UNDERSTANDING AND PROTECTING CLOSED-SOURCE SYSTEMS THROUGH DYNAMIC ANALYSIS

A Thesis
Presented to
The Academic Faculty

by

Brendan Dolan-Gavitt

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2014

UNDERSTANDING AND PROTECTING CLOSED-SOURCE SYSTEMS THROUGH DYNAMIC ANALYSIS

Approved by:

Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Jonathon Giffin
HP Fortify

Patrick Traynor
School of Computer Science
Georgia Institute of Technology

Weidong Cui
Microsoft Research

Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Date Approved: 21 August 2014

To my parents,

Terry Dolan and Phil Gavitt.

ACKNOWLEDGEMENTS

I am deeply grateful to my current advisor, Wenke Lee, for his guidance, support, and understanding as I made my way, however circuitously, through the PhD program. My erstwhile advisor Jon Giffin was an unfailing and invaluable source of insight and stimulating conversation; I will always fondly remember hours-long conversations in my first few years at Georgia Tech on topics ranging from program analysis and algorithms to Buffy the Vampire Slayer. Finally, Patrick Traynor's Destructive Research course my first fall semester helped kickstart my research career, and I have always found his advice and insight valuable. My mentors outside of Georgia Tech have also been crucial to my development: Weidong Cui and David Molnar of Microsoft Research were sources of inspiration and invaluable guidance during my summer in Seattle. I would especially like to thank Tim Leek of MIT Lincoln Laboratory; during the five years I have worked with him he has proved himself to be not only a brilliant collaborator and wise mentor but also a true friend.

My friends in Atlanta, Boston, and elsewhere have made the highs higher and the lows endurable, and I could not ask for a smarter or kinder group of people in my life. My best friends deserve special mention: Emily Wang, my oldest and dearest friend, whose friendship has nourished me through hard and lean times; and Ying Xiao, the most intelligent and learned man I know, who always challenges me to do and be better than I think I can. I also give thanks, in no particular order, to Sarah Montgomery, Randall Munroe, Rosemary Mosco, Patrick Hulin, Anthony Eden, Josh Hodosh, Ryan Whelan, Catherine Grevet, Yacin Nadji, Chris Berlind, Emma Cohen, Sara Krehbiel, Anita Zakrzewska, Prateek Bhakta, and everyone else that deserves a place here but whom I have momentarily forgotten. Your love and friendship has made all the difference these past six years.

Annuska Riedlmayer has, for the past four amazing years, been my love, my partner in adventure, my patient confidant, and my constant companion. Thank you for your patience as I worked on this thesis so far away, for teaching me so much, and for bringing so much

joy into my life.

Thomas Dolan-Gavitt, my brother and friend, talented writer and filmmaker, and all-around wonderful human being, thank you for so many late-night talks, shared ideas, movie and music recommendations, and moral support. I love you, bro.

Finally, I must thank my parents, Terry Dolan and Phil Gavitt, to whom this thesis is dedicated, for their unconditional love and support. Growing up, I never once doubted that they placed the interests of their children first, and I know they made many sacrifices to ensure that we had the best education possible; I hope that my work, in some small measure, can stand as a testament to their efforts.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The modern world is critically dependent on complex software whose internal details are often tenuously understood, even by their creators. In order to properly protect deployed systems whose authors may be unable or unwilling to divulge critical details of their systems' operation, we must develop ways of understanding deployed binary software; furthermore, because the amount of deployed code vastly outstrips our ability to analyze it manually, we need techniques that can automate these analyses. The techniques of program analysis, originally created to help test software in development, have increasingly been used by those in the security community to help understand binary programs without cooperation from code creators. These analyses have typically focused on static analysis of the structure of the code in question; while this can bear significant fruit at small scales, it usually cannot handle large-scale software such as operating system kernels.

In this dissertation, we focus instead on dynamic analyses that examine the data handled by programs and operating systems in order to divine the undocumented constraints and implementation details that determine their behavior in the field. In doing so, we make practical a wide variety of new security applications that previously depended on manual reverse engineering, including RAM forensics and virtual machine introspection.

First, we introduce a novel technique for uncovering the constraints actually used in OS kernels to decide whether a given instance of a kernel data structure is valid. Such information is critical in developing intrusion detection systems and forensic tools that look for hidden and unlinked structures, but previous approaches to finding these constraints relied on unprincipled guesswork. Our system dynamically probes for enforced invariants by mutating the content of existing data structure instances and monitoring the operating system's response to determine whether the modification has degraded system functionality. In doing so we can infer which constraints on the structure's fields are actually enforced.

Next, we further enable progress in secure system monitoring and forensics by tackling the semantic gap problem – the disparity in semantic understanding between what is needed by security systems and what is provided by low-level views (e.g., raw physical memory) of running systems. We present a pair of systems that allow, on the one hand, automatic extraction of whole-system algorithms for collecting information about a running system, and, on the other, the rapid identification of "hook points" within a system or program where security tools can interpose to be notified of events such as file or URL access, cryptographic key generation, and kernel log messages. Together, these bridge the semantic gap by allowing periodic passive monitoring of system state and immediate notification of specific security events, respectively. Each relies on dynamic analyses that start with examples of security-relevant data and identify code within the system that handle or report such data.

Finally, we present and evaluate a new dynamic measure of code similarity that examines the content of the data handled by the code, rather than the syntactic structure of the code itself. This problem has implications both for understanding the capabilities of novel malware as well as understanding large binary code bases such as operating system kernels.

CHAPTER I

INTRODUCTION

1.1 Motivation and Goals

We are surrounded by complex, closed systems every day. These deployed systems are fragile and were often not created with security in mind, but are expensive to replace and will likely be with us for decades. They often include software whose manufacturers have long since gone out of business, and for which no detailed documentation still exists. Even in cases where closed-source software still has vendor support, attitudes toward intellectual property and trade secrets may prevent a vendor from divulging the internal workings of their software.

Given the difficulty of getting code creators to willingly hand over implementation details, how can such systems be effectively protected? The inner workings of a system are of vital interest to security systems: the difference between a simple bug and a major vulnerability quote often hinges on some minor detail of implementation; forensics software must understand undocumented formats and algorithms in order to reconstruct a picture of a suspect's activity that will hold up in court; and intrusion detection and prevention systems must understand the intentionally obscure techniques malware uses to insinuate itself into a system and carry out attacks.

Beyond merely protecting existing systems, there are other security motivations for developing a detailed understanding of the implementation details of closed-source software. It may often be necessary to understand the undocumented behavior of some code without cooperation from its author. In the case of malware, this point is obvious, but even for benign software we may need to peer into its internals to verify that it works as claimed. For example, in 2013, researchers challenged [99] Apple's claims that messages sent by its iMessage chat client could not be read by anyone but the intended recipient (including Apple); in refuting this claim researchers had to reverse engineer the iMessage client. Even

more disturbingly, by reverse engineering the TOM-Skype client (a special version of Skype distributed in China) Knockel et al. [60] discovered a secret list of words that would trigger censorship and surveillance. The question of whether the software one uses is acting in its user's interests can be critical to one's safety and security and can often only be answered through reverse engineering.

For these reasons, in order to protect deployed systems and understand what the software we use is doing, we need automated techniques that can understand software based only on binary artifacts. Although there has been a great deal of research in the past decade on binary software analysis for vulnerability discovery, less work has been done on automatically inferring the details of large scale systems such as operating systems. For such systems, techniques such as static analysis often break down due to the large amount of code and certain constructs prominent in systems code such as indirection through function pointers.

In this work, we seek to develop techniques that can elucidate the internal workings and assumptions of modern operating systems and programs. In particular, our focus will be on techniques which allow security tools to effectively monitor the runtime state and behavior of closed-source systems in order to provide services such as anti-virus monitoring, intrusion detection, and rootkit detection. Our tools of choice will be dynamic analyses, for these typically scale better to large code bases than static analysis, and will use the data handled and output by the system as a guide to code analysis, as it is ultimately data, not code, which determines the state of the system.

1.2 Thesis Overview

Security systems often cannot rely on the assumption that the operating system has not been tampered with; because of this, they must come to their own understanding of the high-level state of the system independent of the provided APIs. To achieve this goal, this thesis presents **novel dynamic analysis techniques for automatically understanding closed-source systems in order to develop better protections**. Our first three chapters focus on the application of such techniques to intrusion detection systems depicted

in Figure 1, where an exteranal security monitor must understand internal details of program and operating system implementation in order to detect threats. Our final chapter discusses a more general approach to program understanding, in which dynamic features of execution are used to classify unknown behaviors in benign and malicious code.

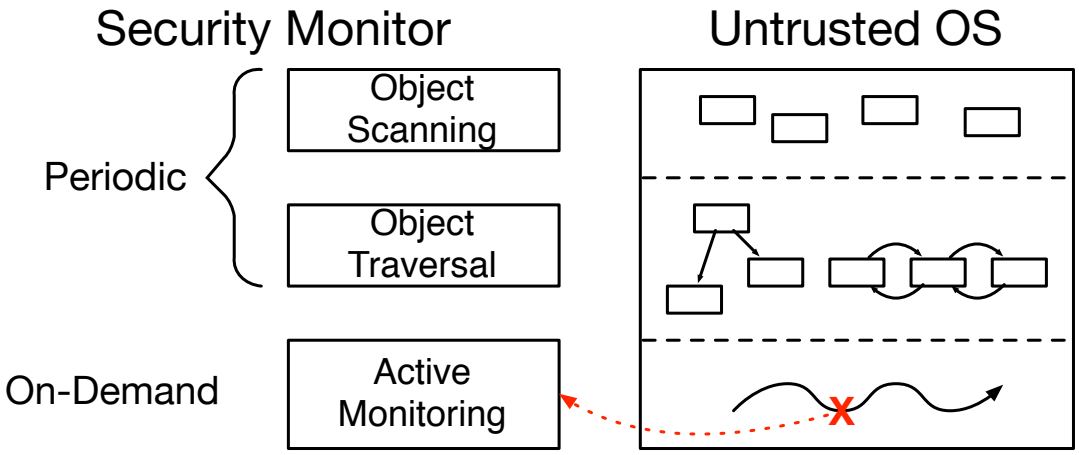


Figure 1: The architecture of an intrusion detection system that attempts to place as little trust in the operating system as possible. Because the OS cannot be trusted, the security monitor must reproduce and understand implementation details that are not normally made public.

In Chapter 2, we examine the use of *data structure signatures* for detecting hidden and unlinked structures in operating system memory. Such signatures rely on the fact that data structure instances typically have *invariants* on their fields that allow them to be located by looking for patterns in memory. These invariants, however, must be enforced by the operating system, or attackers will still be able to hide kernel objects by manipulating the structure's fields in ways that violate the supposed "invariant". Thus, to place memory scanning techniques and develop *robust signatures*, we introduce a dynamic analysis based on memory access profiling and fuzzing that probes data structure fields in order to detect which invariants are actually enforced and can therefore be safely used in signatures. Armed with such signatures, the security monitor in Figure 1 can compare the operating system's self-reported view of the system and what actually exists in memory.

Rather than scanning memory for relevant objects, intrusion detection systems may

instead traverse the data structures that represent the system state in the same way that the operating system or program would—i.e., by starting from some global variables, following pointers that make up linked list or tree structures, and so on. This method, however, relies on much more intimate knowledge of the structures and algorithms used by the system and the burden of reverse engineering is correspondingly higher (this mismatch between the high-level semantic view needed by a security monitor and a low-level view of e.g. physical memory is often referred to as the *semantic gap*). In Chapter 3, we introduce Virtuoso, a system which automatically extracts small subprograms from a larger operating system or application that implement the work traversing the program's data structures and interpreting its internal state. This is done by identifying some API that outputs the desired data and then using *dynamic slicing* and *trace merging* to identify and extract the code responsible for computing that data. In essence, Virtuoso extracts and encapsulates a program or operating system's own mechanisms for interpreting its internal state and allows them to be used independent of the original program. In the IDS shown in Figure 1, these extracted programs can be used for periodic object traversal.

Passive monitoring is only half the story for an intrusion detection system, however. Real-world security systems also need to support *active monitoring*, where events that occur in the system trigger notifications to the security monitor. Such events include file and URL accesses, log messages, cryptographic key generation (so that encrypted network traffic can be monitored), and so on. In order to support this use case, we need ways of quickly locating the code in a system that is responsible for carrying out the actions we are interested in monitoring. In Chapter 4 we describe Tappan Zee (North) Bridge, or TZB, a system that leverages the fact that the data associated with a security event will be written to and read from memory when the event occurs; thus, by monitoring all memory accesses made in the system and searching through these streams for data associated with the event in question, we can locate the code location where a hook should be placed. This information can the be provided to provide the final piece of the intrusion detection system shown in Figure 1, the active monitoring component.

Finally, in Chapter 5 we examine a new technique for for detecting code with similar

functionality by looking at the content handled by functions during a dynamic trace. The problem of identifying functionality in an unknown, binary code base is one that is relevant to several security problems: malware analysts need to quickly determine the capabilities of previously unseen malware and relate them to existing samples; vulnerability researchers often need to quickly zero in on portions of code that are prone to vulnerabilities such as input parsing; and reverse engineering efforts of all sorts can benefit from a high-level view of what different sections of code are likely to do before performing a detailed analysis of any one part.

We conclude in Chapter 6 by examining a number of broad, open problems in this area and considering what changes new systems might incorporate in order to make themselves more amenable to the kind of detailed internal inspection needed for security monitoring.

CHAPTER II

ROBUST SIGNATURES FOR KERNEL DATA STRUCTURES

2.1 Motivation

Many successful malware variants now employ kernel-mode rootkits to hide their presence on an infected system. A number of large botnets such as Storm, Srizbi, and Rustock have used rootkit techniques to avoid detection. This has led to an arms race between malware authors and security researchers, as each side attempts to find new methods of hiding and new detection techniques, respectively. For example, the FU Rootkit [17] introduced a means of process hiding known as Direct Kernel Object Manipulation (DKOM), which unlinks the malicious process from the list of active processes maintained by the system. In response, some forensic memory analysis tools [13, 104, 124] have started scanning kernel memory using signatures for process data structures, and comparing the results with the standard process list. Because signature-based scanning only requires access to physical memory, scanners are most useful in an offline forensic context, but can be used for live analysis as well.

However, a signature-based search can only be effective if it is difficult for an attacker to evade. As Walters and Petroni [125] note, many current signatures for process data structures in particular can easily be fooled by modifying a single bit in the process header. Although this field is normally constant, its value is irrelevant to the correct operation of the process, and so an attacker with the ability to write to kernel memory can easily modify it and evade detection. This leads naturally to the question of which fields in a given data structure are, in fact, essential to its function and would therefore be good features on which to base a signature.

In this chapter, we describe a principled, automated technique for generating robust signatures for kernel data structures such as processes. We employ a feature selection process

that ensures that the features chosen are those that cannot be controlled by an attacker—attempting to evade the signature by modifying the features will cause the operating system to crash or the functionality associated with the object to fail. We use our methods to derive a signature for `EPROCESS`, the data structure used to represent a running process in Windows. By construction, an attacker attempting to evade the signature by altering the fields of the process structure will harm the functionality of the OS or process. In addition, we will show conclusively that current, manually generated signatures are trivially evadable by attackers that can write to kernel memory.

Our feature selection mechanism uses two methods to determine which portions of the data structure are critical to its function. First, we monitor operating system execution and note which fields it reads and writes in the target structure. The intuition is that fields that are never accessed cannot cause a crash if modified by an attacker and hence are poor features for robust signatures. Next, we attempt to determine which fields can be modified by an attacker without preventing the data structure from working correctly. This stage of feature selection simulates the behavior of an attacker attempting to evade a signature: if an attacker can arbitrarily modify a field, then any constraint we devise for that field could be bypassed.

After robust features have been selected, we collect samples of those features in the data structure from a large number of instances in memory images. We then use a dynamic invariant detection technique [35] to find constraints on their values that can be used in a signature. Our signature generator uses these constraints to create a plugin for the Volatility memory analysis framework [124] that can find these data structures in memory.

We demonstrate the advantage of our automatically generated signatures over existing solutions by creating a prototype rootkit (based on FU [17]). This custom malware hides processes using a combination of DKOM and signature evasion techniques. By altering unused fields in the process structure that current signatures depend on, the rootkit successfully evades existing signature-based process scanners. We show that our scanner is capable of detecting processes hidden using this method in Windows memory images.

We chose to apply our technique to the problem of finding processes in Windows memory

images for several reasons. Reliable identification of running programs is a basic security task that is a prerequisite for many other types of analysis. In addition, process hiding is common feature of kernel malware; a single rootkit may be used to hide the presence of wide variety of user-level malware. However, our techniques are general, and we will discuss the possible application of our techniques to other kernel data structures in Section 2.7.

We make the following contributions: first, we provide strong empirical evidence that existing signatures are trivially evadable. Second, we develop a *systematic* method for securely selecting features from a data structure that can be used to create highly robust signatures. Finally, we present a method for generating a signature based on robust features, and use it to create a specific signature for process objects on Windows that is as accurate as existing signatures for current malicious and non-malicious processes, but is resistant to evasion.

These results are of immediate importance to a number of security tools which rely on being able to locate data structures in kernel memory. The virtual machine-based "out-of-the-box" malware detection system proposed by Jiang et al. [52], for example, uses several invariant bytes found in the header of a process structure to find processes under Windows. Cross-view detection approaches to detect hidden processes used by memory analysis tools such as Volatility [124], memparser [13], and PTFinder [104] also make use of signatures to locate key kernel structures. Finally, virtual machine introspection libraries such as XenAccess [94] often use signature scans of guest memory to identify processes and provide user-space address translation. The ability to locate data structures such as processes, independent of any operating system-level view, is critical to the correct operation of these tools, and all of them would benefit from the use of more robust signatures.

2.2 Related Work

Signature-based methods have repeatedly been proposed to identify particular classes of security threats, and, in general, these methods have been found vulnerable to evasion in the face of adversaries. In the area of virus detection, for example, the earliest detectors (and, indeed, many modern commercial utilities) matched byte strings found in viral code

that were unlikely to occur in innocuous programs. As the volume of viral code in the wild increased, automated methods were developed to generate signatures based on known viral samples [57]. These methods were thwarted by the appearance of polymorphic and metamorphic [119] viruses; with these techniques, virus creators could transform the malicious code into a form that was functionally equivalent but had no meaningful strings in common with the original code. As a complexity theory problem, reliable detection of bounded length metamorphic viruses has been shown to be NP-complete [114]. Empirical results have confirmed the difficulty of the problem: by mutating Visual Basic viruses found in the wild using techniques similar to fuzzing [39, 85, 86] and random testing, Christodorescu and Jha [25] found that most malware detectors are vulnerable to even simple obfuscation techniques.

The response to network-based worms followed a similar path. Initial attempts to detect network worms used simple, handmade signatures for intrusion detection systems such as Snort [100] that searched for static byte patterns in the network payload of the worm. However, such manual processes did not scale to the large number of worm variants that soon appeared, and numerous systems for automatic signature generation were proposed [58, 65, 107]. These too, however, were soon defeated by polymorphic shellcode that altered the syntactic structure of the worm payload without affecting its functionality [30]. Although later signature generation systems [73, 92] were able to create signatures based on invariant features in certain classes of polymorphic shellcode, Gundy et al. [45] found that there were some vulnerabilities that could not be captured by such systems. Indeed, further work by Song et al. [113] demonstrated that the general problem of modeling polymorphic shellcode was likely to be infeasible, and Fogla and Lee [38] found that detecting polymorphic blending attacks is an NP-complete problem.

Although these results do not make the search for reliable kernel data structure signatures look promising, there are key differences that allow signature-based methods to be effective in this case. In the case of viruses and shellcode, the syntax of the malicious input is under the control of the attacker; only its semantics must remain the same in order to produce an effective attack. By contrast, *the syntax of kernel data structures is controlled*

by the code of the operating system; an attacker can only modify the data contained in the structure to the extent that the operating system will continue to treat it as a valid instance of the given type. By identifying portions of these data structures that cannot be modified by the attacker, we are able to generate signatures that resist evasion.

Our signature generation system uses dynamic analysis to profile field usage in kernel data structures. A similar technique is employed by Chilimbi et al. [21] for a different goal; their tool, bbcache, analyzes field access patterns in user-space data structures in order to optimize cache performance. After the profiling step, we use a technique similar to fuzzing [85] to identify unused fields in kernel data structures. Fuzzing was also applied by Solar Eclipse [111] to determine which fields in the Portable Executable (PE) file format were required by the Windows loader, in order to develop ways of decreasing the size of Windows executables. Finally, our system finds invariants on the fields in the data structure and produces a Python script that can be used to find instances of the structure in memory images.

Another system that makes use of data structure invariants is Gibraltar by Baliga et al. [7]. Their system creates a graph of all kernel objects in memory and records the values of those objects' members as the system runs. The dynamic invariant detector Daikon [35] is then used to derive constraints on the objects' data. Deviations from the inferred invariants are considered to be attacks against kernel data. The goals and assumptions of our own system, however, are substantially different: whereas Gibraltar assumes that the locations of all kernel data structures can be found *a priori* and then attempts to enforce constraints on those objects, our system seeks to find features of specific data structures in order to locate them in memory. The two approaches can be seen as complementary; once security-critical objects such as processes are located using our signatures, techniques similar to Gibraltar may be employed to detect and enforce invariants on the data.

A number of approaches to detect hidden processes have been featured in other work. Antfarm [53] and Lycosid [54] track the value of the CR3 register as a virtual machine executes to identify unique virtual address spaces, which correspond to distinct processes. Although this approach is quite useful in a live environment, it cannot be used for offline

forensic analysis. Some offline methods have been proposed as well: Klister [102], for example, attempted to find hidden processes by relying on the scheduler's thread list rather than the systemwide process list, which thwarts some kinds of DKOM attacks. An evasion for this kind of detection has been demonstrated [2], however: an attacker can replace the OS scheduler with a modified copy, bypassing any tool which relies on the original list. Signature scanning is less vulnerable to such attacks, as any changes an attacker makes to the layout of a data structure must be reflected in any OS code that uses the structure.

Finally, other recent work has focused on finding data structures in memory. Laika [27] infers the layout of data structures and attempts to find instances in memory using unsupervised Bayesian learning. Because their system assumes that the data structures are not known in advance, it may be useful in cases where data structure definitions are not available. This flexibility comes the cost of accuracy, however: whereas our process scanner found all instances of the structure in all tested memory images with no false positives, Laika had false positive and negative rates of 32% and 35%, respectively.

We anticipate that rootkit authors will soon add signature evasion techniques to their standard toolkits. Evasion of signatures for kernel data has already been publicly discussed: Walters and Petroni [125] demonstrated that changing a single bit in the Windows process data structure was sufficient to evade all known signatures without harming the functionality of the running process. Similarly, in response to Rutkowska's signature-based modGREPER [103], valerino [120] described an evasion technique that altered a number of fields in the driver and module structures. Finally, bugcheck [15] described a number of methods (including signatures that match fixed strings) for finding kernel data structures in Windows memory and explored several evasion techniques that could be used to hide objects from those techniques. As tools that find hidden objects through memory scans become more common, we believe malware authors will adapt by attempting to evade signatures. This threat motivates our work to generate signatures for kernel data that are resistant to evasion.

Type == 0x03
Size == 0x1b
ThreadListHead >= 0x80000000
DirectoryTableBase is aligned to 0x20

Figure 2: A naïve signature for the `EPROCESS` data structure. The constraints shown are a subset of those used in PTFinder's process signature. Because the `Size` field is not used by the operating system, an attacker can change its value, hiding the process from a scanner using this signature.

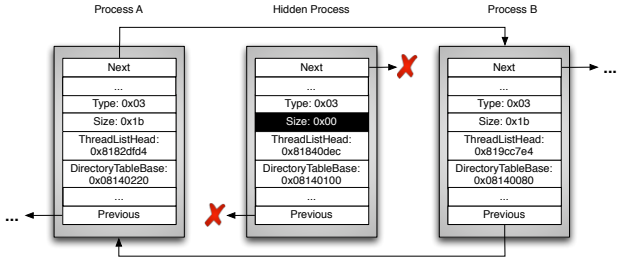


Figure 3: A portion of the process list while a process hiding attack is underway. The hidden process has been removed from the doubly linked list, and its `Size` field has been changed to evade the signature above.

2.3 Overview

A signature-based scanner examines each offset in physical memory, looking for predefined patterns in the data that indicate the presence of a particular data structure. These patterns take the form of a set of *constraints* on the values of various fields in the structure. For example, Figure 2 shows a simple signature for a process data structure in Windows (called `EPROCESS`; this structure holds accounting information and metadata about a task). The constraints check to see that the `Type` and `Size` fields match predefined constants, that the `ThreadListHead` pointer is greater than a certain value, and that the `DirectoryTableBase` is aligned to 0x20 bytes. These invariants must be general enough to encompass all instances of the data structure, but specific enough to avoid matching random data in memory.

An adversary's goal is to hide the existence of a kernel data structure from a signature-based scanner while continuing to make use of that data structure. We assume that the attacker has the ability to run code in kernel-mode, can read and modify any kernel data, and cannot alter existing kernel code. This threat model represents a realistic attacker: it

is increasingly common for malware to gain the ability to execute code in kernel mode [37], and there are a number of solutions available that can detect and prevent modifications to the core kernel code [97,105], but we are not aware of any solutions that protect kernel data from malicious modification.

To carry out a process hiding attack, such as the one shown in Figure 3, an attacker conceals the process from the operating system using a technique such as DKOM. This attack works by removing the kernel data structure (`EPROCESS`) representing that process from the OS's list of running processes. The process continues to run, as its threads are known to the scheduler, but it will not be visible to the user in the task manager. However, it will still be visible to a signature-based scanner [13,104] that searches kernel memory for process structures rather than walking the OS process list.

To evade such scanners, the attacker must modify one of the fields in the process data structure so that some constraint used by the signature no longer holds. The field must also be carefully chosen so that the modification does not cause the OS to crash or the malicious program to stop working. In the example shown in Figure 3, the attacker zeroes the `Size` field, which has no effect on the execution of his malicious process, but which effectively hides the data structure from the scanner.

In order to defend against these kinds of attacks, signatures for data structures must be based on invariants that the attacker cannot violate without crashing the OS or causing the malicious program to stop working. The signature's constraints, then, should be placed only on those fields of the data structure that are critical to the correct operation of the operating system. Rather than relying on human judgement, which is prone to errors, our solution profiles OS execution in order to determine the most frequently accessed fields, and then actively tries to modify their contents to determine which are critical to the correct functioning of the system. Such fields will be difficult for an attacker to modify without crashing the system, and are good candidates for robust signatures.

Finally, we will demonstrate that it is possible to automatically infer invariants on these robust fields and construct a scanner that is resistant to evasion attacks.

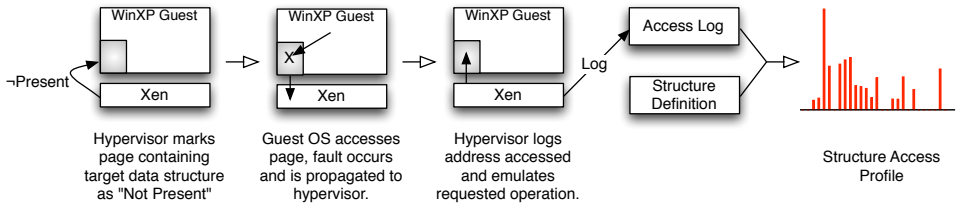


Figure 4: The profiling stage of our signature generation system. As the OS executes, accesses to the target data structure are logged. These logs are then combined with knowledge of the field layout in the structure to produce an access profile.

2.4 Architecture

Our system architecture generates signatures using a three step process. We first *profile* the data structure we wish to model to determine which fields are most commonly accessed by the operating system (Section 2.4.1). This is done to narrow the domain of the data that we must test in the fuzzing stage: if a field is never accessed in the course of the normal operation of the OS, it is safe to assume that it can be modified without adversely affecting OS functionality. Next, the most frequently accessed fields are *fuzzed* (Section 2.4.2) to determine which can be modified without causing a crash or otherwise preventing the structure from serving its intended purpose. Finally, we collect known-good instances of the data structure, and build a signature based on these instances that depends only on the features that could not be safely modified during fuzzing (Section 2.4.3).

Profiling and fuzzing are both essentially forms of *feature selection*. Each tests features of the data structure to determine their suitability for use in a signature. Features that are unused by the operating system or are modifiable without negative consequences are assumed to be under the control of the attacker and eliminated from consideration. Including such weak features would allow an attacker to introduce *false negatives* by violating the constraints of a signature that used them, in the same way that a polymorphic virus evades an overly specific antivirus signature. At the other end of the spectrum, if too few features remain at the end of feature selection, the resulting signature may not be specific enough and may match random data, creating *false positives*.

The profiling and fuzzing stages are implemented using the Xen hypervisor [9] and VMware Server [123], respectively. Because profiling requires the ability to monitor memory

access, we chose to use Xen, which is open source and allowed us to make the necessary changes to the hypervisor to support this monitoring. However, Xen lacks the ability to save and restore system snapshots, a feature needed for reliable fuzz testing, so we use VMware Server for this stage. Also, because VMware's snapshots save the contents of physical memory to disk, we were able to easily modify the memory of the guest OS by altering the on-disk snapshot file.

2.4.1 Data Structure Profiling

In the profiling stage (shown in Figure 4), we attempt to determine which structure fields are most commonly accessed by the operating system during normal operation. Fields which are accessed by the OS frequently are stronger candidates for use in a signature because it is more likely that correct behavior of the system depends upon them. By contrast, fields which are rarely accessed are most likely available to the attacker for arbitrary modification; if the OS never accesses a particular field in the data structure, its value cannot influence the flow of execution.

In our implementation, we make use of a modified Xen hypervisor and the "stealth breakpoint" technique described by Vasudevan and Yerraballi [121] to profile access to the data structure. Stealth breakpoints on memory regions work by marking the memory page that contains the data to be monitored as "not present" by clearing the Present bit in the corresponding page table entry. When the guest OS makes any access to the page, the page fault handler is triggered, an event which can be caught by the hypervisor. The hypervisor then logs the virtual address that was accessed (available in the `CR2` register), emulates the instruction that caused the fault, and allows the guest to continue. These logs can later be examined to determine what fields were accessed, and how often.

For example, to monitor the fields of the Windows `EPROCESS` data structure, we launch a process and determine the address in memory of the structure. We then instruct the hypervisor to log all access to that page, and then allow the process to run for some time. Finally, the logs are examined and matched against the structure's definition to determine how often individual fields were read or written. This process is repeated using several

different applications; only the fields that are accessed during the execution of *every* program will be used as input for the fuzzing stage.

We note in passing that determining the precise field accessed requires access to the data structure's definition. On open source operating systems, this information is easy to come by, but for closed source OSes such as Windows it may be more difficult to obtain. For our implementation, which targets Windows XP, we used the debugging symbols provided by Microsoft; these symbols include structure definitions for many kernel structures, including `EPROCESS`.

2.4.2 Fuzzing

Although a field that is accessed frequently is a stronger candidate than one which is never accessed, this condition alone is not sufficient to identify truly robust features for use in signatures. For example, the operating system may update a field representing a performance counter quite frequently, but its value is not significant to the correct operation of the OS. Thus, to be confident that a signature based on a particular field will be robust against evasion attacks, we must ensure that the field cannot be arbitrarily modified.

The actual fuzzing (shown in Figure 5) is done by running the target operating system inside VMware Server [123]. As in the profiling stage, we first create a valid instance of the data structure. Next, the state of the guest VM is saved so that it can be easily restored after each test. For each field, we then replace its contents with test data from one of several classes:

1. **Zero**: null bytes. This is used because zero is often a significant special case; e.g., many functions check if a pointer is NULL before dereferencing.

2. **Random**: n random bytes from a uniform distribution, where n is the size of the field.

3. **Random primitive type**: a random value appropriate to the given primitive type. In particular, pointer fields are fuzzed using valid pointers to kernel memory.

4. **Random aggregate type**: a random value appropriate to the given aggregate type

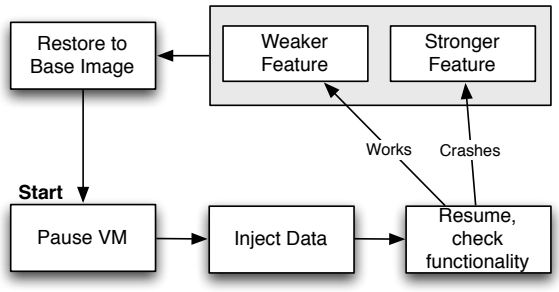


Figure 5: The architecture of the fuzzing stage. Starting from some baseline state, a test pattern is written into a field in the target data structure in the memory of the virtual machine. The VM is then resumed and tested to see if its functionality is intact. If so, the modification was not harmful and we consider the field a weaker candidate for a signature.

(i.e., structure). Embedded structures are replaced by other valid instances of that structure, and pointers to structures of a given type are replaced by pointers to that same type. Currently implemented as a random choice of value from that field in other instances of the target data structure.

After the data is modified, we resume the guest VM and monitor the operating system to observe the effects of our modifications. To determine whether our modification was harmful, we must construct a test, ϕ, which examines the guest and checks if the OS is still working and the functionality associated with the target data structure instance is still intact.

For a process data structure, ϕ could be a test that first checks to see if the OS is running, and then determines whether the associated program is still running. This check may be simpler to program if the behavior of the application in question is well-known. In our experiments (described in Section 2.5.1), we used an application we had written that performed some simple functions such as creating a file on the hard drive. This allowed ϕ to check if the program had continued to run successfully by simply testing for the existence of the file created by the program.

To actually inject the data, we pause the virtual machine, which (in VMware) writes the memory out to a file. We then use the memory analysis framework Volatility [124] (which we modified to support writing to the image) to locate the target instance and modify the appropriate field with our random data. Volatility was ideal for this purpose, because it has

the ability to locate a number of kernel data structures in images of Windows memory and provides a way to access the data inside the structures by field name. The modifications to allow writing to the image (a feature not normally supported by Volatility, as it is primarily a forensics tool) required 303 lines of additional code.

Finally, we resume the virtual machine and check to see if ϕ indicates the system is still functioning correctly after some time interval. This interval is currently set to 30 seconds to allow time for the VM to resume and any crashes to occur. Software engineering studies [90] have found that crashes typically occur within an average of one million instructions from the occurrence of memory corruption; thus, given current CPU speeds, it is reasonable to assume that this 30 second delay will usually be sufficient to determine whether the alteration was harmful to program functionality. The result of the test is logged, and we restore the saved virtual machine state before running the next test. Any fields whose modification consistently caused ϕ to indicate failure are used to generate a signature for the data structure.

2.4.3 Signature Generation

The final signature generation step is performed using a simplified version of dynamic invariant detection [35]. For each field identified by the feature selection as robust, we first gather a large number of representative values from all instances of the target data structure in our corpus of memory images. Then, for each field, we test several constraint templates to see if any produce invariants that apply to all known values of that field. The templates checked are:

- **Zero subset**: check if there is a subset of the values that is zero. If so, ignore these values for the remaining checks.
- **Constant**: check if the field takes on a constant value.
- **Bitwise AND**: check if performing a bitwise AND of all values results in a non-zero value. This effectively checks whether all values have any bits in common.

```
class Scan(RobustPsScanner,
           PoolScanProcessFast2.Scan):
    def __init__(self, poffset, outer):
        RobustPsScanner.__init__(self,
            poffset, outer)
        self.add_constraint(
            self.check_objecttable
        )
        self.add_constraint(
            self.check_grantedaccess
        )

    [...]

    def check_objecttable(self, buf, off):
        val = read_obj_from_buf(buf,
            types, ['_EPROCESS',
            'ObjectTable'], off)
        res = (val == 0 or
            (val & 0xe0000000 ==
                    0xe0000000 and
            val % 0x8 == 0))
        return res

    def check_grantedaccess(self, buf, off):
        val = read_obj_from_buf(buf,
            types, '_EPROCESS',
            'GrantedAccess'], off)
        res = val & 0x1f07fb == 0x1f07fb
        return res
```

Figure 6: Two sample constraints found by our signature generator. If all constraints match for a given data buffer, the plugin will report that the corresponding location in memory contains an `EPROCESS` instance.

- **Alignment**: check if there is a power of two (other than 1) on which all values are aligned.

First, because many fields use zero as a special case to indicate that the field is not in use, we check if any of the instances are zero, and then remove these from the set to be examined. Constraints are then inferred on the remaining fields, and zero will be included as a disjunctive (OR) case in the final signature. The other templates will produce conjunctive constraints on the non-zero field values.

The *constant* template determines whether a field always takes on a particular value. This is useful, for example, for data structures that have a "magic" number that identifies them uniquely. Because the features that are used for signature generation are known to be robust (as these were the selection criteria described in Sections 2.4.1 and 2.4.2), we can have some confidence that the operating system performs sanity checking on such constant values.

The two remaining tests are particularly useful for finding constraints on pointer values. The *bitwise AND* test simply performs a bitwise AND of all values observed. In many operating systems, kernel memory resides in a specific portion of virtual address space, such as (in Windows) the upper 2 gigabytes. One can determine if a 32-bit pointer always points to kernel memory, then, by simply checking that the highest bit is set.

Finally, the *alignment* test attempts to find a natural alignment for all values. As an optimization, many OS memory managers allocate memory aligned on a natural processor boundary, such as eight bytes. As a result, most pointers to kernel objects will likewise have some natural alignment that we can discover and use as a constraint.

Our signature generator takes as input a comma-separated file, where each row gives the field name and a list of values observed for that field. For each field, it applies the constraint templates to the values listed and determines a boolean expression that is true for every value. It then outputs a plugin for Volatility, written in Python, that can be used to scan for the target data structure in a memory image. An excerpt of the plugin generated to search for instances of the `EPROCESS` data structure is given in Figure 6.

The signature generation mechanism produces extremely robust results in practice: as we describe in Section 2.6.3, the signature we generated for Windows process data structures found all instances of the data structure in memory with no false positives or negatives. Should this technique prove insufficient for some data structure, however (for example, if only a few features are robust enough to use in a signature), more heavyweight techniques such as dynamic heap type inference [98] could be used.

2.4.4 Discussion

Our experiments (described in Section 2.6) show that these techniques can be used to derive highly accurate signatures for kernel data structures that are simultaneously difficult to evade. There are, however, certain drawbacks to using probabilistic methods such as dynamic analysis and fuzz testing. In particular, both techniques may suffer from *coverage* problems. In the profiling stage, it is highly unlikely that every field used by the operating system will actually be accessed; there are many fields that may only be used in special cases.

Likewise, during fuzzing, it is possible that although the operating system did not crash during the 30 seconds of testing, it might fail later on, or in some special circumstances.

In both of these cases, however, we note that these omissions will only cause us to ignore potentially robust features, rather than accidentally including weak ones. Moreover, from an attacker's point of view, the malware need not work perfectly, or run in every special case: sacrificing correct operation in a tiny fraction of configurations may be worth the increased stealth afforded by modifying these fields. Thus, a short time interval for testing is *conservative*: it will never cause a weak feature to be used in a signature, as only features whose modification consistently causes OS crashes form the basis of signatures. However, it may cause fields to be eliminated that would, in fact, have been acceptable to use in a signature. If too many fields are eliminated, the resulting signature may match random data in memory, creating false positives. In any case, this limitation is easily overcome by increasing the amount of time the fuzzer waits before testing the OS functionality, or by exercising the guest OS more strenuously during that time period.

However, there are some coverage issues that could result in weak signatures. Because fuzzing is a dynamic process, it is possible to only inject a subset of values that causes the OS to crash, while there exists some other set of values that can be used without any negative effects. In this case, we may conclude that a given feature is robust when in fact the attacker can modify it at will. For most fields it is not practical to test every possible value (for example, assuming each test takes only five seconds, it would still require over 680 years to exhaustively test a 32-bit integer). In Section 2.8, we will consider future enhancements to the fuzzing stage that may improve coverage.

Finally, we note that although the features selected using our method are likely to be difficult to modify, there is no guarantee that they will be usable in a signature. For example, although our testing found that the field containing the process ID is difficult to modify, it could still be any value, and examining a large number of process IDs will not turn up any constraints on the value. In practice, though, we found that most of the "robust" features identified were fairly simple to incorporate into a signature, and we expect that this will be true for most data structures.

2.5 Methodology

Signature search is essentially a classification problem: given an unknown piece of data, we wish to classify it as an instance of our data type or as something else. Our experiments, therefore, attempt to measure the performance of the signatures using standard classification metrics: false positives and negatives. A false positive in this case is a piece of data that matches the signature but would not be considered a valid instance of the data structure by the operating system. Conversely, a false negative is a valid instance that is not classified as such by our signature. False negatives represent cases where the attacker could successfully evade the signature, whereas false positives could introduce noise and make it difficult for to tell what processes are actually running.

For our purposes, we only consider false positives that are syntactically invalid. We note that an attacker could generate any number of false positives by simply making copies of existing kernel data structures. These structures would be semantically invalid (the operating system would have no knowledge of their existence), but would be detected by a signature scanner. The possibility of such "dummy" structures is a known weakness of signature-based methods for finding kernel data structures [125]; however, a solution to this problem is outside the scope of this work.

For our experiments, we chose to generate a signature for the Windows `EPROCESS` data structure, which holds information related to each running process. This structure was chosen because it is the most commonly hidden by malicious software, and there are a number of existing signature-based tools that attempt to locate this data structure in memory [13, 104, 124]. We compare the success of our signature with these tools. However, our work can also be applied to generate signatures for other data structures.

2.5.1 Profile Generation and Fuzzing

During the profiling stage, we examined access patterns for fields in the `EPROCESS` data structure. To ensure that our data represented a wide range of possible application-level behavior, we chose twenty different programs that performed a variety of tasks (see Table 1 for a full list). To obtain a profile, we first launched the application and noted the address of

Table 1: List of applications profiled, along with the number of fields in `EPROCESS` accessed.

System Utilities		
Name	Version	Fields
Telnet	5.1.2600.5512	112
Command shell	5.1.2600.5512	135
NTFS Defragment	5.1.2600.5512	123
Explorer	6.0.2900.5512	143
Browsers		
Name	Version	Fields
Internet Explorer	7.0.5730.13	153
Mozilla Firefox	3.0.5	147
Games		
Name	Version	Fields
WinQuake	1.06	129
Minesweeper	5.1.2600.0	108
Editor		
Name	Version	Fields
Notepad	5.6.2600.5512	151
Debugger		
Name	Version	Fields
Notepad (debugged)	5.6.2600.5512	145
Windbg (debugging)	6.9.0003.113 x86	146
Communications		
Name	Version	Fields
Outlook Express	6.00.2900.5512	148
Pidgin	2.5.3	143
Installer		
Name	Version	Fields
Pidgin Installer	2.4.0	188
Antivirus / Antispyware		
Name	Version	Fields
Avira AntiVir	8.2.0.337	130
Spybot Search & Destroy	1.6.0.0	136
Network Servers		
Name	Version	Fields
Apache HTTPd	2.2.11	108
network_listener	N/A	139
Multimedia		
Name	Version	Fields
Windows Media Player	9.00.00.4503	142
iTunes	8.0.2.20	142

its associated `EPROCESS` structure using the kernel debugger, WinDbg. We then instructed the Xen hypervisor to monitor access to the page, and used the application for a minimum of five minutes.

We note that in addition to differences caused by the unique function performed by each application, other activities occurring on the system may cause different parts of the data structure to be exercised. In an attempt to isolate the effects caused by differences in program behavior, as each profile was generated we also used the system to launch several new tasks (Notepad and the command shell, `cmd.exe`), switch between interactive programs, and move and minimize the window of the application being profiled.

After profiling the applications, we picked only the features that were accessed in all twenty applications. This choice is conservative: if there are applications which do not cause a particular field to be exercised, then it may be possible for an attacker to design a program that never causes the OS to access that field. The attacker would then be able to modify the field's value at will and evade any signature that used constraints on its value.

As described in Section 2.4.2, features that were accessed by all programs profiled were fuzzed to ensure that they were difficult to modify. Checking that the `EPROCESS` data structure is still functioning after each fuzz test is much simpler if the associated program has known, well defined behavior. For this reason, we chose to create a program called `network_listener` that opens a network socket on TCP port 31337, waits for a connection, creates a file on the hard drive, and finally exits successfully. The baseline snapshot was taken just after launching `network_listener` inside the guest VM.

Because the program behavior is known in advance, the test to see if the OS and program are still working correctly (ϕ) becomes simple. From the host, we perform the following tests on the virtual machine:

1. Determine if the virtual machine responds to pings.
2. Check that the program is still accepting connections on port 31337.
3. Check for the existence of the file written by the application (using the VMware Tools API).

If all tests pass, then ϕ returns true, indicating that the modification was accomplished without harming OS or program functionality. If instead ϕ returns false, then the OS has

crashed or some aspect of the program associated with our `EPROCESS` instance has stopped functioning. This latter case indicates that the OS will not accept arbitrary values for the field, and provides evidence that we can safely build a signature based on the field.

2.5.2 Signature Generation and Evaluation

Finally, we generated a signature using the method described in Section 2.4.3. The features chosen were the 15 most robust, as measured by the tests done during the fuzzing stage. For each of these fields, we extracted from our corpus of memory images (our training set) a list of the values it contained for all processes found in the image. The four images in the training set were not infected by malware, and were taken from systems running the 32-bit version of Windows XP, Service Pack 2. Processes were located in the memory image by walking the operating system's process list; in the absence of maliciously hidden processes, this serves as "ground truth" for the list of valid process data structures. We then used our signature generator to find constraints on the observed values. The signature generator outputs a plugin for Volatility that can be used to search for a data structure matching the constraints found in a memory image.

The generated scan plugin was used to search for processes in a number of memory images. For this purpose, we used two images provided by the NIST Computer Forensic Reference Data Sets (CFReDS) project [91] and a paused virtual machine on which a process had been hidden by our own custom rootkit. The number of false positives and negatives were measured for each test image, and compared against two existing signature-based tools, PTFinder [104] and Volatility's `psscan2` module.[1]

Our custom malware, which is a slightly modified version of the FU Rootkit [17], hides processes using DKOM (as in the original FU), and additionally attempts to evade known process signatures by zeroing non-essential fields in the process data structure. The fields modified, shown in Table 2, were chosen by finding those fields that were used by common scanners but that our initial structure profiling indicated were unused by the OS.

[1]Note that Volatility also includes a process scanner called `psscan`. This scanner uses the same constraints as PTFinder, and hence is vulnerable to the same evasions, so we do not consider it here.

Table 2: Fields zeroed by our modified FU Rootkit, along with the scanners that depend on that field.

Field	Used by
ThreadListHead.Blink	Volatility (psscan2)
Pcb.Header.Type	PTFinder
Pcb.Header.Size	PTFinder
WorkingSetLock.Header.Type	PTFinder
WorkingSetLock.Header.Size	PTFinder
AddressCreationLock.Header.Type	PTFinder
AddressCreationLock.Header.Size	PTFinder

2.6 Results

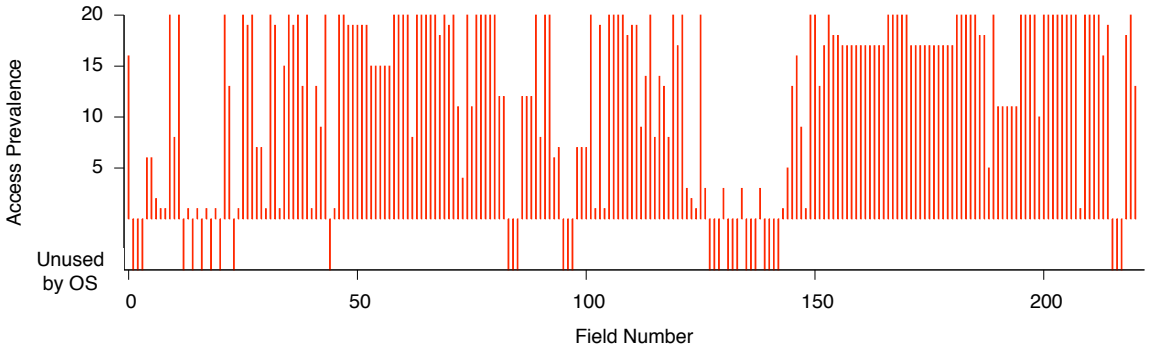
The experimental results are given below. We describe the outcome of profiling twenty different applications and present the results of the fuzzing stage. These features are used by the signature generator to find constraints and create a new process scan module for Volatility. Finally, we compare the accuracy of our scanner with other popular scanners.

Throughout, we also consider what our results tell us about the features used by another popular signature (PTFinder's signature for `EPROCESS`). We find that after fuzzing and profiling, only two of its nine features are resistant to evasion; the remaining invariants are not sufficient to avoid matching random portions of memory.

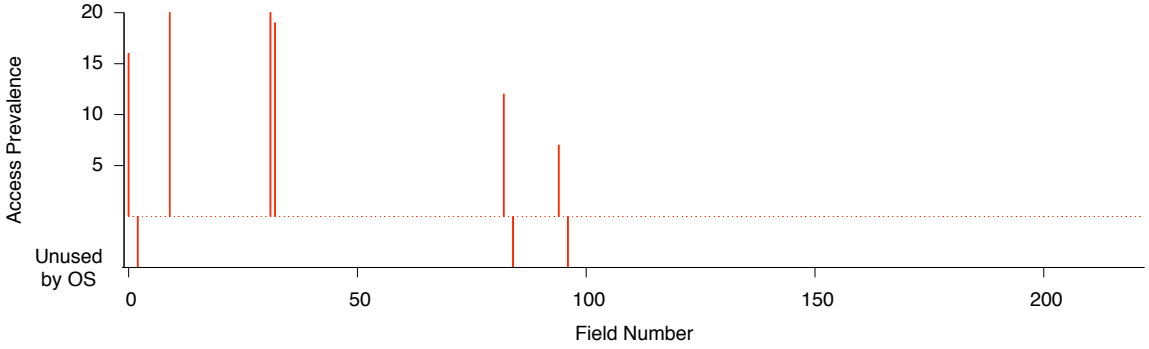
2.6.1 Profiling

After profiling the twenty applications described in Section 2.5.1, we can confirm our hypothesis that some fields are accessed only rarely, if ever. Of the 221 fields in the `EPROCESS` data structure, 32 were never accessed during the execution of the profiled programs. At the other extreme, 72 were accessed for every application and are thus strong candidates for a process signature. In between are 117 fields that were accessed by some programs, but not others; Figure 7a gives a histogram detailing precisely how many programs accessed each field.

Included in the 32 fields that were never accessed are three of the nine used by PTFinder to locate processes in memory dumps; a further four are only accessed by a subset of the programs profiled (the profiling results for the fields used by PTFinder are shown in Figure 7b). Because the signature used in PTFinder is conjunctive (all of its constraints must



(a) Number of profiled programs in which `EPROCESS` fields were accessed. Only fields accessed by all 20 programs provide the strongest assurance for use in a signature.



(b) Access prevalence of fields used by PTFinder's `EPROCESS` signature. Note that the signature relies on field values never used by the OS, so an attacker can safely change these values to evade the signature.

Figure 7: Access prevalence for `EPROCESS` for profiled applications.

be met in order to report a match), and the attacker has complete control over three of the fields used in the signature, *we can conclude that this signature can be trivially evaded.* The features chosen by PTFinder's author did not correspond to those used by the OS, demonstrating that human judgment may not be sufficient to determine what fields are appropriate for use in data structure signatures.

2.6.2 Fuzzing

We then took the 72 fields identified as always accessed during the profiling stage and fuzzed them using the four different data patterns (zero, random, random primitive, and random aggregate), modifying each field with each pattern five times, for a total of 1,440 distinct tests. The overall number of failed tests for each field is shown in Figure 8. However, this does not provide a full picture of the fuzzing results, as it is also important to note which

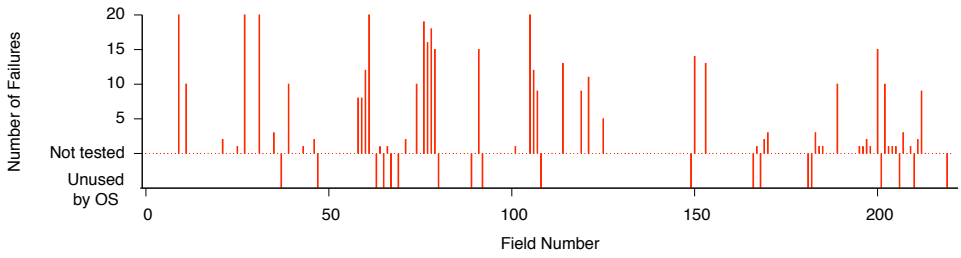


Figure 8: Fuzzing results for `EPROCESS`. The y-axis represents the total number of tests for which ϕ returned false, indicating that the process was no longer functioning correctly. Higher bars indicate stronger features.

data patterns caused the OS to fail. It may be acceptable to use a field in a signature even if it is possible to write zeroes to that field, for example, because the constraint could include zero as a special case. We have, therefore, included several sample fields in Table 3, in order to give an idea of what the result data looks like.

We find, as expected, that there are many "essential" fields upon which we may base our signature. Five fields failed every attempt at manipulation, and a further 12 failed more than half of the tests (i.e., more than 10). This will give us a set of robust features that is large enough to ensure that the number of false positives found by the signature is minimized.

As in the profiling stage, we also note that these results give us a very strong indication of what fields *not* to choose. Of the 72 fields from profiling, 29 passed every test (their modification did not result in any loss of functionality); these, again, would be a poor basis for a signature as their values can be controlled by an attacker.

2.6.3 Signature Accuracy

With a list of robust features in hand, we used our signature generator to find constraints on the values of each feature. The field values were collected from 184 processes across the four images in our training set and constraints were inferred using the templates described in Section 2.5.2, producing the constraints shown in Table 4. The signature generator produced a plugin for Volatility that uses the constraints found to search for `EPROCESS` instances in memory images.

Table 3: Selected `EPROCESS` fields and the results of fuzzing them. The values indicate the number of times a given test caused ϕ to return false, indicating that the OS or program had stopped working correctly. The columns indicate number of OS crashes when testing with the **Z**ero, **R**andom, Random **P**rimitive, and Random **A**ggregate patterns.

Field	**Z**	**R**	**P**	**A**	**Total**
ActiveProcessLinks.Flink	5	5	5	5	20
Pcb.DirectoryTableBase[0]	5	5	5	5	20
Pcb.ThreadListHead.Flink	5	5	5	5	20
Token.Value	5	5	5	3	18
Token.Object	5	5	5	1	16
VadHint	5	5	2	0	12
UniqueProcessId	1	5	5	1	12

We then evaluated the accuracy of three process scanners: our own automatically generated scanner, Volatility's `psscan2` module, and PTFinder [104]. Using each scanner, we searched for instances of the `EPROCESS` data structure in the three memory images listed in Section 2.5.2. The output of each tool was compared against the OS's list of running processes (found by walking the linked list of process data structures using Volatility's `pslist` module). In the case of the non-NIST image, we also checked for the presence of our hidden process, which was not visible in the standard OS process list.

We found that all three scanners had equal detection performance on the NIST images and found every process data structure with no false positives. However, only our own scanner was able to detect the hidden process in the third image, demonstrating that an attacker could potentially evade both psscan2 and PTFinder with minimal effort. We believe our signature will also prove resistant to evasion against real-world attackers, as the features it uses are demonstrably difficult for an attacker to alter.

Aside from the active processes in the images, we also noted some discrepancies between the three scanners with respect to terminated processes whose `EPROCESS` structure was still in memory and had not yet been overwritten. Although PTFinder and psscan2 were vulnerable to the evasion by our custom malware, they also found these terminated processes, which our scanner missed.

As terminated processes could be of forensic interest, we checked whether there was some

Table 4: Constraints found for "robust" fields in the EPROCESS data structure. The operators shown have the same meaning as in C; % stands for the mod operation, and & represents bitwise AND. && and || are the boolean operators for "and" and "or", respectively.

Field	**Constraint**
Pcb.ReadyListHead.Flink	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
Pcb.ThreadListHead.Flink	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
WorkingSetLock.Count	`val == 1 && val & 0x1 == 0x1`
Vm.VmWorkingSetList	`val & 0xc0003000 == 0xc0003000 && val % 0x1000 == 0`
VadRoot	`val == 0 || (val & 0x80000000 == 0x80000000 && val % 0x8 == 0)`
Token.Value	`val & 0xe0000000 == 0xe0000000`
AddressCreationLock.Count	`val == 1 && val & 0x1 == 0x1`
VadHint	`val == 0 || (val & 0x80000000 == 0x80000000 && val % 0x8 == 0)`
Token.Object	`val & 0xe0000000 == 0xe0000000`
QuotaBlock	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
ObjectTable	`val == 0 || (val & 0xe0000000 == 0xe0000000 && val % 0x8 == 0)`
GrantedAccess	`val & 0x1f07fb == 0x1f07fb`
ActiveProcessLinks.Flink	`val & 0x80000000 == 0x80000000 && val % 0x8 == 0`
Peb	`val == 0 || (val & 0x7ffd0000 == 0x7ffd0000 && val % 0x1000 == 0)`
Pcb.DirectoryTableBase.0	`val % 0x20 == 0`

subset of our "robust" features that would find such processes without introducing false positives. By modifying our scanner to report the result of each constraint for the terminated processes, we found that Windows appears to zero the `Token.Object` and `Token.Value` fields, which refer to the process's security token, when the process exits. Once we removed these constraints from our signature, we were able to find the terminated processes reported by other tools without introducing false positives. We note that our scanner remains resistant to evasions, as the remaining fields are all robust. The terminated processes demonstrate the importance of generating signatures from a training set that represents the full range of objects one wishes to detect.

2.7 Other Structures

Although our experiments have only been run on `EPROCESS`, we are confident that the technique will generalize to other data structures. Certain structures in particular, such as threads (represented by `ETHREAD` in Windows) and files (`FILE_OBJECT`), would be good candidates for signature generation, as they contain a wealth of information about the runtime state of the system that is useful for forensic analysis. We will briefly consider

what changes might be needed to generate signatures for these structures.

The profiling stage is essentially the same for any structure: the objects are created by some user-level program (i.e., by spawning a thread or opening a file), their location in memory is determined, and the memory region is monitored to log access to the structure. In the fuzzing stage, the only significant challenge is creating an appropriate functionality test ϕ. As threads contain executable code, one could simply use the same test as for processes: attempt to create a file and ensure that the file is created successfully. For file objects, one could test functionality by performing a range of operations on the open file, such as reading, writing, seeking, and closing the file. Finally, our signature generator is not specific to any one object type and could be used as-is: the only input required is a list of observed values for each field in the data structure.

One final complication may arise if the target structure is fairly small. In this case, it may be that after eliminating weak features, there will not be enough left to create a reliable signature (in the sense of having few false positives). In this case, we might employ a more sophisticated search technique: rather than simply using basic pattern matching to find instances of the structure, we could take advantage of information such as the types of objects to which it points. This technique has previously been used successfully in other work to identify the types of objects on the heap [98], and this additional contextual information could improve signature accuracy.

2.8 Future Work

As discussed in Section 2.4.4, obtaining full coverage during fuzzing is impractical; however, but it may be possible to improve our coverage through more judicious selection of random data. For example, we might incorporate *mutation fuzzing* [93], which generates fuzz data by creating small, random variations on existing values. This would help us more efficiently explore the space of possible values, as for many fields legal values will be clustered fairly close together.

The profiling stage could also be made more accurate by switching from simply monitoring whether a field is accessed to attempting to determine how it is used. This would

involve the use of taint tracking [11] to find out whether the value of a given field actually influences the execution of the OS. We expect that this could significantly reduce the number of fields that would need to be fuzzed.

Finally, although the automatically generated signatures from our method appear to work well, they are based on dynamic analysis and may therefore suffer from coverage problems. Gaps in coverage could lead to false negatives and evasions in the signature matching process: a constraint inferred on a small number of samples may not be representative of the full range of values that field uses, and thus be overly restrictive. To improve confidence in such constraints, one could also use static analysis to attempt to prove that the inferred constraints do, indeed, hold in all cases.

2.9 Conclusions

We have successfully demonstrated that it is possible to automatically select robust features of data structures and generate evasion-resistant signatures based on them. More importantly, we have described a systematic way of determining which features to use when creating a data structure signature. To our knowledge, no such method was previously available, and we believe that many applications will benefit from this technique.

Our work resulted in a new signature for process data structures on Windows, which can be used immediately by applications which require the ability to locate processes in memory. We also showed that existing signatures used by memory analysis applications were vulnerable to evasion, and in the case of PTFinder we described precisely which constraints could be violated by an attacker. These concrete contributions significantly increase the difficulty of hiding process objects from signature scans on Windows systems.

CHAPTER III

VIRTUOSO: NARROWING THE SEMANTIC GAP IN VIRTUAL MACHINE INTROSPECTION

3.1 Motivation

As kernel-level malware has become both more common and more sophisticated, some researchers have advocated protecting insecure commodity operating systems through the use of virtualization-based security [42, 55, 69, 95]. By moving protection below the level of the OS, these solutions aim to provide many traditional security services such as intrusion detection and policy enforcement without fear of subversion by malicious code running on the system. At the same time, because the code base of the hypervisor is small, there is some hope that it can be verified and provide secure isolation for the virtual machines it supervises.

This enhanced security and isolation comes at a cost, however. Because the guest operating system is untrusted, information about its state can not be reliably obtained by calling standard APIs within the guest. Instead, security tools must use *introspection* to retrieve information about the current state of the guest OS by examining the physical memory of the running virtual machine and using detailed knowledge of the operating system's algorithms and data structures to rebuild higher level information such as lists of running processes, open files, and active network connections. Even if the guest OS is compromised, the tools using introspection will continue to report accurate results, so long as the underlying internal data structures used by the operating system are intact.

The creation and maintenance of introspection tools, therefore, is dependent on detailed, up-to-date information about the internal workings of commodity operating systems. Even for systems where the source code to the kernel is available, acquiring this knowledge can be a daunting task; when source is not available, prolonged effort by a skilled reverse engineer may be required. Moreover, the time and effort spent divining the internals of a particular

version of an operating system may not be applicable to future versions. This problem of extracting high-level semantic information from low-level data sources, known as the *semantic gap*, presents a significant barrier to the development and widespread deployment of virtualization security products.

The fact that creating introspection tools by hand is difficult has multiple security implications. Security professionals and system administrators rely upon the output of hand-built introspection tools; at the same time, these tools are fragile and often cease to function upon application of operating system patches. Currently, therefore, one must often choose between keeping an up-to-date system and maintaining a working set of introspections. Moreover, hand-built introspection tools may introduce opportunities for attackers to evade security tools: to the extent that introspection must replicate the behavior of in-guest functionality, the developer must have an accurate model of the OS's inner workings. Garfinkel [41] has demonstrated, with numerous examples drawn from his experience with system call monitors, that any divergence between reality and the model assumed by a security program can provide opportunities for evasion.

In this chapter, we both ease the burden of maintaining introspection-based security tools and make them more secure by providing techniques to *automatically* create programs that can extract security-relevant information from outside the guest virtual machine. Our fundamental insight is that it is typically trivial to write programs inside the guest OS that compute the desired information by querying the built-in APIs. For example, to get the ID of the currently running process in Linux, one need only call `getpid()`; by contrast, implementing the same functionality from outside the virtual machine depends on intimate knowledge of the layout and location of kernel structures such as the `task_struct`, as well as information about the layout of kernel memory and global variables. Our solution, named Virtuoso, takes advantage of the OS's own knowledge (as encoded by the instructions it executes in response to a given query) to learn how to perform OS introspections. For example, using Virtuoso, a programmer can write a program that calls `getpid()` and let Virtuoso trace the execution of this program, automatically identify the instructions necessary for finding the currently running process, and finally generate the corresponding introspection

code.

Virtuoso takes into account complexities that arise from translating a program into a form that can run outside of its native environment. It must be able to extract the program code, and its dependencies; to do this, we use a dynamic analysis that captures *all* the code executed while the program is running. At the same time, Virtuoso must be able to distinguish between the introspection code and unrelated system activity, a challenge that we overcome by making use of dynamic slicing [63] to identify the exact set of instructions required to compute the introspection. Finally, the use of dynamic analysis means that a single execution of the program may not cover all paths required for reliable re-execution. To ensure that our generated programs are reliable, we introduce a method for merging multiple dynamic traces into a single coherent program and show that its use allows our generated programs to achieve high reliability.

3.2 Related Work

Most closely related to our own techniques are two recent works which perform extraction of binary code for security purposes. Binary Code Reutilization (BCR) [18] and Inspector Gadget [62] focus on the problem of extracting subsets of binary code, particularly in order to reuse pieces of malware functionality (such as domain name generation or encryption functions). While the former primarily uses static analysis and recursively descends into function calls, the latter adopts a dynamic approach similar to our own. Inspector Gadget performs a dynamic slice on a log collected from a single run of the malicious code and changes half-covered conditional branches into unconditional jumps (the authors do not discuss the possible correctness issues that may arise from this strategy, nor do they say whether their system can combine multiple dynamic traces into a single program). Both systems are more limited in scope than Virtuoso and do not consider kernel code. They make extensive use of domain knowledge about the target operating system, and would therefore be difficult to adapt to new platforms. By contrast, Virtuoso uses no domain knowledge aside from understanding the x86 platform, and performs *whole-system* binary code extraction.

More generally, virtual machine introspection has played an integral role in a number of recent security designs. Garfinkel and Rosenblum [42] first proposed the idea of performing intrusion detection from outside the virtual machine. Their system introspects on Linux guests to detect certain kinds of attacks; higher level information is reconstructed through the use of the `crash` utility [87]. Since then, other researchers have proposed using virtualized environments for tasks such as malware analysis [31, 52] and secure application-layer firewalls [116]; these systems all make heavy use of virtual machine introspection and therefore must maintain detailed and accurate information on the internals of the operating systems on which they introspect.

Introspection can also be useful in contexts outside of traditional virtualization security. Petroni et al. [96] presented a system called Copilot that polls physical memory from a PCI card to detect intrusions; such detection needs introspection to be useful. Malware analysis platforms that run samples in a sandboxed environment, such as CWSandbox [128] and Anubis [10], can also benefit from introspection: by extracting high-level information about the state of the system as the malware runs, more meaningful descriptions of its behavior can be generated. At the same time, this introspection must be secure and unobtrusive, as the guest OS is guaranteed to be compromised. Our work can help provide higher-level semantic information to such systems.

Originally proposed by Korel and Laski [63], the dynamic slicing at the heart of our algorithm has been used in several other recent security tools, such as HookMap [127] and K-Tracer [68]. The former makes use of this technique to identify memory locations in the kernel that could be used by a rootkit to divert control flow, while the latter uses a combination of dynamic backward and forward slicing (also known as "chopping") to analyze the behavior of kernel malware with respect to sensitive data such as process listings. Additionally, Kolbitsch et al. [61] described a malware detection system that uses dynamic slicing to extract the code necessary to relate return values and arguments between system calls.

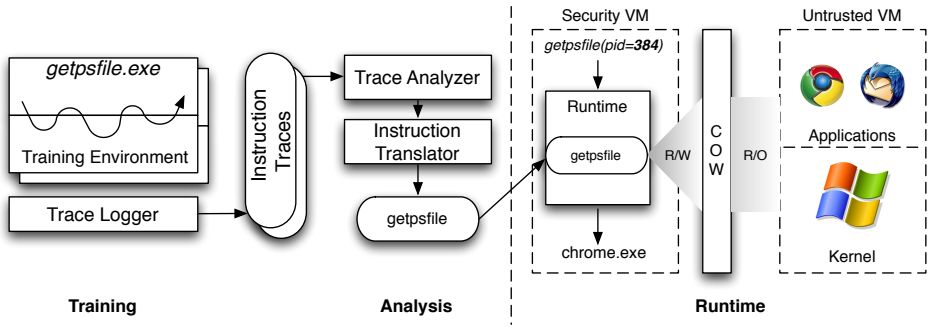


Figure 9: An example of Virtuoso's usage. A small in-guest *training program* that retrieves the name of a process given its PID is executed in the training environment a number of times, producing several instruction traces. These traces are then analyzed and merged to create an introspection program that can be used to securely introspect on the state of a guest virtual machine from outside the VM. We explain further details of this diagram in Section 3.4.

3.3 Introspection for Security

To motivate the design of Virtuoso, we consider the common scenario where *secure introspection* into the state of a running virtual machine (known as the *guest*) from outside that VM (i.e., from the *host*) is needed. This need may arise when performing out-of-the-box malware analysis or debugging [52,55], or when attempting to secure a commodity operating system by placing security software in an isolated virtual machine for intrusion detection [42] or active monitoring [95]. In cases such as these it is easy to see why in-guest monitoring is undesirable: if the guest OS can be compromised, then the in-guest tools themselves can become corrupted or the system APIs can be altered to return false information (such as by omitting a malicious process from a task list).

Developing tools to perform such introspections, however, is no easy task: extracting even simple information such as the current process ID or a list of active network connections requires deep understanding of the operating system running in the VM. In particular, the *location* of the data, its *layout*, and what *algorithms* should be used to read it must all be known. While some recent work has addressed the problem of automatically reverse engineering data structure layouts [70, 77, 109], the overall task of creating introspections is still largely a manual effort. With Virtuoso, however, another option is available: by simply writing small programs (which we refer to as *training programs*) for the guest OS that

extract the desired information, host-level introspection tools can be *automatically* created that later retrieve the same information at runtime from outside the running VM.

To illustrate how Virtuoso can be used to quickly create secure introspection programs, consider the running example shown in Figure 9. Here, a developer wishes to create an introspection that obtains the name of a process by its PID. To do so, he first writes an in-guest program that queries the OS's APIs for the information. Such programs are generally trivial to create for even modestly skilled programmers. The program is then run inside Virtuoso, which records multiple executions, analyzes the resulting instruction traces, and creates an out-of-guest introspection program that can retrieve the name of a process running inside a virtual machine. This program can then be deployed to a secure virtual machine and used to obtain this security-relevant information at runtime. The generated program does not call any in-guest APIs, but rather contains all the low-level instructions necessary to implement that API outside of the virtual machine. This entire procedure involves only minimal effort on the part of the developer, and requires no reverse engineering or specialized knowledge of the OS's internals.

Tools generated in this way are specific to the operating system for which the corresponding training programs were developed; that is, if the developer writes a program that gets the current process ID in Windows XP, the generated out-of-the-box tool will only work for Windows XP guests. However, supporting a new platform only requires adapting the training program to a new OS and its API, rather than costly reverse engineering or source code analysis by a specialist. Moreover, different releases of various operating systems are often backwards compatible at either the source or binary level. This is the case, for example, with Microsoft Windows: programs written for Windows NT 3.1 (released over 15 years ago) can run without modification on Windows 7. In this case Virtuoso could analyze the behavior of the same program under each release of Windows, generating a suite of host-level introspection tools to cover any version desired with no additional programmer effort. Changes in the underlying implementations of the OS APIs would be automatically reflected in the generated host introspection tools.

Once the introspection tools have been generated for a particular OS release, they can

be deployed to aid in the secure monitoring of any number of virtual machines running that OS. The generated introspection tools will provide accurate data about the current state of the guest VM, even when the code of that VM is compromised.[1] Thus, Virtuoso produces tools that enable secure monitoring of insecure commodity operating systems, and accomplishes this task with only minimal human intervention.

3.3.1 Scope and Assumptions

Although we believe Virtuoso represents a large step forward in narrowing the semantic gap, there are some fundamental limitations to our techniques and constructs that Virtuoso is not currently equipped to handle. First, Virtuoso is, by design, only able to produce introspection code for data that is already accessible via some system API in the guest. These introspections are a subset of those that could be programmed manually, and it is possible that some security-relevant data are not exposed through system APIs. For those functions that are exposed through system APIs, Virtuoso greatly reduces the cost of producing introspections; we describe six such useful introspections in Section 3.6.

Second, there are certain code constructs that are difficult to reproduce, such as synchronization primitives (which may require other threads to act before an introspection can proceed) and interprocess communication (IPC). These limitations are not neccessarily fundamental, but they require additional research to overcome. We discuss these limitations further in Section 3.7.

Finally, if an introspection requires interaction with a hardware device, such as the disk or a network interface, the introspection cannot currently be generated by Virtuoso. In Section 3.7 we discuss possible extensions to Virtuoso that could enable automatic generation of some such introspections, however, for some devices any interaction might irreversibly perturb the guest state (for example, if an introspection must access the network, there is no way to reverse its effects—the packets cannot be "unsent"). This would leave the guest in an inconsistent state and may cause the system to become unstable; it also violates the principle that introspection should not alter the guest.

[1] As with all introspection, malicious alterations to the guest's kernel data structure layouts, locations, or algorithms may still cause incorrect results to be reported.

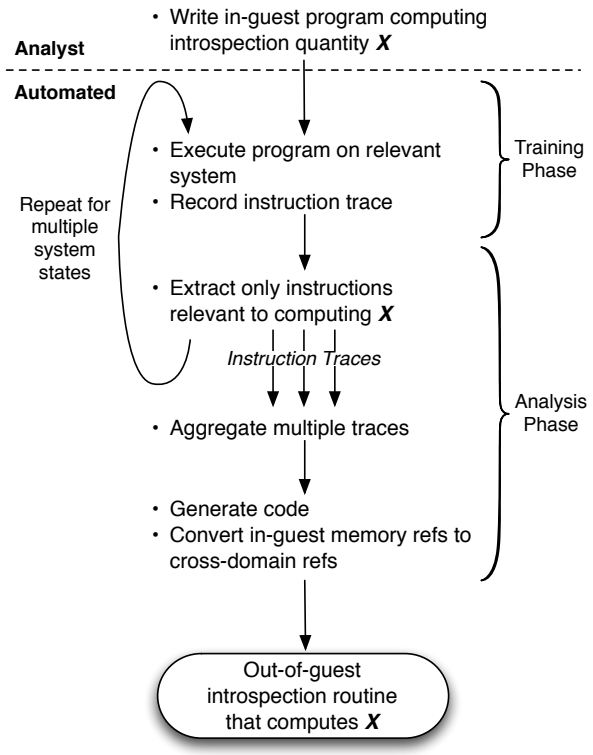


Figure 10: A high-level conceptual view of our system for generating introspection tools. This view corresponds to the training and analysis phases shown in Figure 9. The programmer creates an in-guest program that computes the desired introspection quantity by calling standard OS APIs. This in-guest program is then run repeatedly under various system states, and the instructions executed are logged. These instruction traces are then analyzed to isolate just the instructions that compute the introspection quantity, merged into a unified program, and then translated into an out-of-guest introspection tool.

3.4 Virtuoso Design

At a high level, Virtuoso creates introspection tools for an operating system by converting in-guest programs that query public APIs into out-of-guest programs that reproduce the behavior of the in-guest utilities. This is done in three phases (Figure 10): a *training phase*, in which an in-guest program that computes the desired information is run a number of times, and a record of each execution is saved; an *analysis phase*, which extracts just the instructions involved in computing the data required for the introspection, merges the traces into a single program, and translates that program into one that can be executed from outside the virtual machine; and finally a *runtime phase*, in which the generated program is used to introspect on a running virtual machine. These steps are described in more detail

```
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include "vmnotify.h"

int main(int argc, char **argv) {
  char *name;
  HANDLE h;
  DWORD pid = atoi(argv[1]);
  name = (char *) malloc(1024);
  vm_mark_buf_in(&pid, 4);
  vm_mark_buf_in(&name, 4);
  h = OpenProcess(PROCESS_QUERY_INFORMATION,0,pid);
  GetProcessImageFileName(h,name,1024);
  vm_mark_buf_out(name, 1024);
  return 0;
}
```

Figure 11: A complete sample program that gets the name of a process given its PID under Windows. Programs such as these are easy to write from inside the operating system using documented APIs. `GetProcessImageFileName`, `OpenProcess`, and `malloc` are standard API calls known to Windows programmers. `vm_mark_buf_in` and `out` notify Virtuoso of input and output.

below.

3.4.1 Training

The training phase is shown in Figure 9, our running example: the training program, `getpsfile.exe`, is run several times in the training environment to capture a variety of program paths, producing a number of instruction traces. For this stage, we assume the developer can create such small, in-guest programs that compute the data needed for the introspection tool. This places a very minimal burden on the programmer: because ordinary programs running inside the target operating system need to query many of the same kinds of information desired by introspection tools, OS designers typically make rich APIs available for finding out this information. Additionally, such APIs are generally easy to use, stable, and well-documented, as they are the primary interface through which programs talk to the OS.

A sample program that retrieves the name of a process under Windows is shown in Figure 11. The program, annotated with markers that indicate where to begin and end tracing (`vm_mark_buf_in` and `out`), invokes the Windows API functions `OpenProcess` and `GetProcessImageFileName` to get the process name. The annotations also record the addresses of input and output buffers for the program; these will ground the data flow analysis

performed to help identify what portions of the system's execution are necessary to compute the introspection quantity (in this case, the process name).

Even for simple programs like the one described above, there may be a number of different execution paths taken through the program depending on the state of the OS when the program is executed. To produce a program that works correctly in all situations, we run the program repeatedly in an instrumented environment and collect traces that record all instructions executed by the program and operating system. These execution traces are then analyzed and merged together to produce an out-of-guest introspection tool that can be used to reliably provide security-relevant information for monitoring applications.

3.4.2 Analysis

Although the execution traces gathered in the training phase contain all the instructions needed to compute the introspection quantity, they also contain a large amount of extraneous computation. This extra "noise" is present because our traces record the execution of the whole system: in addition to computing the introspection quantity, the traces also contain unrelated events such as interrupt handling and unrelated housekeeping tasks performed by the kernel. Because we are only interested in the code that is necessary for producing a working introspection program, we must excise these extraneous parts of the traces. We accomplish this by first throwing away or replacing parts of each trace that we know *a priori* to be unrelated to the introspection, such as hardware interrupts and memory management. Next, we identify the instructions that actually compute the output by performing a dynamic data slice [63] on each trace. Finally, we merge the slice results across basic blocks and traces, producing a unified program that can be translated into an out-of-guest introspection routine.

In order to produce a working routine, the inputs and outputs to the training program must be identified. To do so, we look for places in the trace where buffers marked as inputs are used. These instructions are marked as places where input data is used so that, at translation time, we can generate code that splices in the program inputs at these points.

Similarly, as the location of the output buffer may change depending on the current environment (for example, if the output buffer is on the stack, its location will change depending on the value of `ESP` when the introspection tool is run), the analysis determines which instructions are immediately responsible for writing the data to the output buffer. These "output-producing instructions" are marked; the translator will handle such instructions specially, so that the output data can be found independent of the runtime CPU state.

Once the instruction trace has undergone slicing, merging, input splicing and output instruction marking, the merged traces are translated into an out-of-guest introspection program. The programs generated by Virtuoso consist of a set of basic blocks together with successor information. Within a basic block, each individual instruction is implemented by a small snippet of code in a high-level language. The use of a high-level language allows our generated programs to be usable in many different contexts: forensic memory analysis, virtual machine introspection, and low-artifact malware analysis.

In our example (Figure 9), we can see that the Trace Analyzer and Instruction Translator together comprise the analysis phase, and create the final introspection program `getpsfile` from the instruction traces generated during training.

3.4.3 Runtime Environment

The translated code, however, cannot be executed directly. Instead, it must be provided with an appropriate *runtime environment* in which to execute. The runtime environment (shown on the right in Figure 9), is installed on the host and runs the translated instructions in a context that gives them access to the resources (such as CPU registers and memory) of the guest virtual machine without perturbing its state.

In particular, low-level operations that affect registers or memory are wrapped so that they enforce *copy-on-write (COW)* behavior. Whenever a write to memory occurs, it is redirected to the COW memory space (shown in the right panel of Figure 9); when a read occurs, it first checks whether the data exists in COW memory. If so, it reads from the COW space. Otherwise, the read is serviced directly from guest memory. A similar scheme is used to handle register access. This allows introspections to access the state of the running

system without interfering with its operation.

There is some subtlety to the question of how to obtain values from the memory of the guest. Certain data seen during training may be specific to the training program (for example, static data such as strings), while other data may be part of the system's state as a whole. In the former case, we would want to use the values seen in training (as they may not be available in the environment in which the introspection program is run), whereas in the latter we would want to obtain the values from the guest itself (since the point of introspection, after all, is to obtain the state of the system). To differentiate between these two types of data, we currently make the simplifying assumption that *kernel* data seen during training is global, and should be read from the guest, while *userland* data is specific to the training program, and should be saved during training and re-used at runtime. This reflects the nature of the introspections we wish to capture, which obtain information about the state of the system as a whole rather than any individual process.

Although this policy currently limits the type of introspections that can be generated to those that inspect systemwide state, one could define other policies that make introspection into specific processes possible. If Virtuoso were provided with some means of locating the appropriate process context at runtime, one could define a policy that reads all data directly from the guest, allowing introspection into the memory of a specific process.

When performing an introspection (even with hand-generated tools), the guest CPU is paused in order to prevent the contents of memory from changing during the introspection. It might seem that one could keep the CPU running, improving performance by allowing the introspection to be carried out in parallel with the guest's execution. This performance boost would come at the cost of reliability, however, as changes to the underlying data examined by the introspection tool would almost certainly cause it to crash or report incorrect results. In addition, we have noticed that most generated introspections must be run when the CPU is in user mode. This is because the traces are recorded based on a userland program, and so some assumptions may be made about the state of global registers in userland that do not hold in kernel mode. To ensure reliability, our runtime environment polls for a

user-mode CPU state and pauses the guest before performing any introspection.[2]

Finally, the Instruction Translator has special handling for instructions marked by the Trace Analyzer as inputs or outputs. Instructions that require input are replaced with equivalent instructions that explicitly take input from the environment in which the introspection tool runs. Output-producing instructions are translated so that their output is placed in a special buffer at a known location; when the program terminates, the contents of this buffer containing the desired introspection quantity are returned so they may be used by security tools. In our running example (Figure 9), we can see that the generated introspection program is provided with the PID 384 as input, and produces the output "chrome.exe".

Note that the contents of the output may still need to be interpreted (for example, by decoding the binary data as a string or integer, or even more complex interpretations such as displaying a timestamp in human-readable form). We provide a basic facility by which the user can provide a function that interprets the output from the runtime environment; however, we do not attempt to include this functionality in the generated introspection tool itself. We feel that this approach is justified, as the output will be the same as that produced by APIs within the guest OS, which we have assumed are well-documented. Therefore, the format of their outputs will most likely also be documented by the OS vendor.

3.5 Implementation

Having given a high-level view of our system, we now turn to the details of its implementation. Virtuoso's trace logging portion is built on top of QEMU [12], a fast emulator and binary translator. Our Trace Analyzer is written in Python, and consists of 1,843 SLOC;[3] the majority of this code consists of the data flow transfer functions for each QEMU micro-instruction. Finally, the Instruction Translator and runtime environment are 431 and 1,095 lines of Python code, respectively. These components, and their logical relationship, are

[2]Rather than polling, it is also possible to set up a callback within the guest whenever there is a transition from kernel to user mode, using the techniques described by Dinaburg et al. [31].

[3]Source line counts were generated using David A. Wheeler's 'SLOCCount'.

Original x86 instruction:

```
MOV EDI, DWORD PTR [EBP+0x10]
```

QEMU micro-operations:

```
MOVL A0,  EBP
ADDL A0,  0x10
LDL  T0,  A0      ; 0x1b1acf00
MOVL EDI, T0
```

Figure 12: An example of an x86 instruction and the corresponding logged micro-instructions. The implicit pointer dereference has been made explicit, and the referenced address has been recorded to enable data flow analysis.

shown in Figure 9 on page 37. We will discuss in detail the implementation of each component in this section.

3.5.1 Trace Logging

Our trace logger, which is shown on the left in Figure 9, is a modified version of QEMU. QEMU works by dynamically translating each guest instruction to a series of operations in a micro-instruction language. The micro-instructions are implemented by small C functions, which are compiled to host-level code and chained together to emulate the guest code on the host.

To enable logging, we inserted small logging statements into each micro-instruction's implementation. These logging statements record the current operation, along with its operands and any dynamic information needed to enable the data flow analysis (see Section 3.4.2 for more details). Thus, when the guest code is translated by QEMU, the generated code will be interleaved with logging functions that record the micro-instructions executed. This interleaving is necessary because accurate logging depends on dynamic information that will only be available once the previous instruction has finished executing.

Our logging functions record the current micro-operation, its parameters, and concrete *physical* memory addresses for each load and store instruction. Using the physical address allows data flow to be reconstructed even when multiple virtual addresses reference the same physical memory (this often occurs when a buffer is shared between a user application and

```
#define MAGIC_IN  0xdeadbeef

void vm_mark_buf_in(void *buf, int len) {
  void *b = buf;
  int n = len;

  // [ register save omitted ]
  __asm MOV EAX, 0
  __asm MOV EBX, MAGIC_IN
  __asm MOV ECX, b
  __asm MOV EDX, n
  __asm CPUID
  // [ register restore omitted ]
}
```

Figure 13: In-guest code that signals the Trace Logger to begin tracing. Code that simply saves and restores clobbered registers has been omitted.

the kernel, for example). Finally, we record the page table dependencies for each memory operation. This is necessary because each memory load or store implicitly depends on the addressing structures used: if another instruction in the trace has modified the virtual to physical address map, that instruction must also be included in order for the resulting program to function correctly. By recording the address of the page directory and page table entries used in calculating the address of the load or store, our dynamic slicing algorithm (described in Section 3.5.3) can automatically determine what page table-manipulating code must also be included.

In Figure 12, we give an example of an x86 instruction and the corresponding QEMU micro-operations that are recorded by the Trace Logger. The x86 instruction is decomposed into a series of simpler operations: first, a memory address is computed by taking the value in the `EBP` register and adding `0x10` to it. Next, the `LDL` micro-op retrieves the value from memory at that address and stores it in the temporary register `T0`. Along with the micro-op, the logger records the address the memory was read from to capture any data flow dependencies. Finally, the value read is transferred to the `EDI` register.

These micro-instructions provide a natural IR for our analyzer. Not only are QEMU micro-ops simple to log, they also satisfy the requirement of performing relatively simple operations with few side effects. This greatly simplifies the task of tracking data flow, as the data defined and used by each micro-instruction is generally easy to understand.

Inside the guest, the *training program* (described in Section 3.4.1) must be able to signal

the Trace Logger and inform it of the beginning and end of the introspection operation, as well as the location of the buffer where the output has been placed. Our implementation accomplishes this by co-opting the x86 `CPUID` instruction. When a magic value is placed in the `EBX` register, our modified QEMU will interpret `CPUID` as a signal to start or stop logging, and will record the values of `ECX` and `EDX` as the buffer's start and size, respectively. This allows us to bound our trace so that it includes only the operations relevant to introspection, and also provides the Trace Analyzer with the locations of the input and output buffers. The in-guest code that implements the notification is given in Figure 13.

3.5.2 Preprocessing

As discussed in Section 3.4.2, the trace is preprocessed before slicing and translation in order to remove hardware interrupts and replace calls to memory-allocating functions with function summaries (this is part of the analysis phase, i.e., the middle portion of Figure 9). Filtering interrupts has two benefits: it reduces the amount of code we need to analyze, and it eliminates many accesses to hardware devices (which may not be available at run time). Replacing memory allocation functions (such as malloc and realloc), while not necessary in theory, is currently required by our implementation, as portions of the OS-provided memory allocation facilities are carried out in the page fault handler, and we do not support hardware exceptions in our runtime environment.

Interrupt filtering is performed in the obvious way, by matching up pairs of interrupts (which are logged by the Trace Logger) and `iret` instructions (which are used in the x86 architecture to return from an interrupt). Because interrupts may be nested arbitrarily deep, we use a counter to ensure that we have matched a given `iret` with the appropriate interrupt. Once the outermost interrupt and `iret` have been identified, they can be excised from the trace.

There are two exceptions to interrupt filtering, however. First, *software interrupts* (i.e., those triggered by an `int` instruction) are part of the code of the program and are treated in much the same way as a standard function call. Second, the x86 architecture, starting with the Pentium, includes an Advanced Programmable Interrupt Controller (APIC) unit,

```
GET_ARG 2
MALLOC          ; placeholder
MOVL A0, ESP
LDL T0, A0      ; 0x1e7beb4
ADDL ESP, 0x10
JMP T0
```

Figure 14: A malloc replacement that summarizes the effect of `RtlAllocateHeap` in Windows using QEMU micro-ops. The summary reads the size of the allocation from the stack, has a placeholder op for the actual allocation, reads the return address, cleans arguments from the stack, and executes the return.

which gives the operating system more control over how and when it receives interrupts. Among the capabilities provided by the APIC is the ability to send interrupts to other processors (including itself). When combined with the ability to defer interrupts by setting the Task Priority Register (TPR) on the APIC, this provides a mechanism that can be used by the OS to perform *asynchronous procedure calls*. Because these are effectively software interrupts (they originate in OS code), we include them in our analysis and in the generated program.

Finally, we provide replacements for the memory allocation functions of some operating systems. To do so, we make use of three pieces of domain knowledge about the OS being analyzed: the virtual address of the function, the number of argument bytes (to enable callee-cleanup, if the calling convention requires it), and the position of the "size" argument relative to the stack. With this information, one can edit the trace and replace a call to (e.g.) malloc with a summary that allocates memory on the host. A summary for RtlAllocateHeap on Windows is shown in Figure 14. We currently replace `RtlAllocateHeap`/`RtlFreeHeap` [83] on Windows, and `malloc`/`realloc`/`calloc`/`free` on Linux. In our testing, programs generated for the Haiku operating system did not need malloc summaries.

3.5.3 Dynamic Slicing and Trace Merging

After preprocessing the trace, our Trace Analyzer uses *executable dynamic slicing* [63] to trace the flow of information through the program, starting with the output buffer specified in the log. For a detailed discussion of the algorithm, refer to Korel and Laski [63];

Algorithm 1 Trace Merging Algorithm. P is the final, merged program, which consists of a number of *block*s and corresponding successors. T is the set of traces. Each trace $t \in T$ consists of a sequence of *op*s. $slice[t]$ is the subset of *op*s in t that are needed, initialized with a dynamic data slice on the output. $dyn\_slice()$ is an instantiation of the dynamic slicing algorithm, and $find\_block()$ identifies a basic block to which an *op* belongs.

```
1: slices ← ∅, blocks ← ∅
2: for t ∈ T do
3:    slice[t] ← dyn_slice(t, t.output)
4:    blocks[t] ← split_basic_blocks(t)
5: end for
6: repeat
7:    P.blocks ← ∅, P.succs ← ∅
8:    for t ∈ T do
9:       for op ∈ slice[t] do
10:         op_block ← find_block(blocks[t], op)
11:         if (op_block ∈ P.blocks) then
12:            p_op_block ← find_block(P.blocks, op)
13:            p_op_block.add(op)
14:         else
15:            P.succs[op_block] ← op_block.succs
16:            P.blocks ← P.blocks ∪ new_block(op)
17:         end if
18:      end for
19:   end for
20:   for block ∈ P.blocks do
21:      if P.succs[block].size > 1 then
22:         conds ← find_exit_conds(block)
23:         for t ∈ T, block ∈ t do
24:            br_ops ← dyn_slice(t, uses(conds))
25:            slice[t] ← slice[t] ∪ br_ops
26:         end for
27:      end if
28:   end for
29:   for t ∈ T do
30:      for op ∈ slice[t] do
31:         for t′ ∈ T, t′ ≠ t, op ∈ t′ do
32:            slice[t′] ← slice[t′] ∪ op ∪ dyn_slice(t′, uses(op))
33:         end for
34:      end for
35:   end for
36: until nothing changed
```

however, we will summarize the idea here. Note that because we do not have access to the full program, our dynamic slicing algorithm cannot calculate full control dependency information as in the original dynamic slicing algorithm. Instead, we include *every* control

flow statement (and its dependencies) that was observed to have more than one successor in the dynamic control flow graph. This is safe (in the program analysis sense of the word), but may over-approximate the dynamic slice.

Dynamic slicing works backward through the trace, looking for instructions that define the desired output data. For each instruction, the set of data *defined* by the instruction is examined. If the data it defines overlaps with the working set, then the instruction handles tracked data, so we must add it to the slice. The working set is then updated by removing the data defined by the instruction and adding the data *used* by the instruction. The algorithm terminates when top of the trace is reached. The output, an *executable dynamic slice*, is precisely the sequence of instructions used to compute the data in the output buffer.

Virtuoso builds the final introspection program from a training set of more than one trace. This is crucial to achieving reliability: except in the case of fairly trivial introspections, a single trace is unlikely to include enough code to adequately cover all important functionality. Consider the fact that many of the kinds of introspections we want to support, such as enumerating running processes or loaded modules, involve traversing collections such as linked lists or arrays. Typically, the code that accesses these data structures includes logic (and therefore multiple paths through the program) to cover corner cases, such as when the collection is empty. If we want our final introspection tool to be able to handle these uncommon (but hardly rare) situations correctly, we must build our program from multiple traces. In practice, to ensure reliability, we generate increasingly reliable programs iteratively: after generating an initial program, it is tested on a variety of system states, and the failing test cases are then used as new training examples. Although this cycle of testing and training is currently done manually, it could easily be automated.

The trace merging algorithm works as follows. First, a merged control-flow graph is constructed from the control flow graphs of the individual traces (lines 8–19 in Algorithm 1). Second (lines 20–28), for each block which has more than one successor (meaning there was a branch or a dynamic jump), a slice is performed on the data neccessary to compute the branch condition (i.e., x86 status and condition flags or the dynamically computed jump target); this step ensures that both loops and branches are faithfully reproduced in the

generated program. Third, Virtuoso performs a *slice closure* (lines 29–35): if an op is in some but not all of the slices, then it is added to the other slices, and a new dynamic slice is performed on the dependencies of that op (this has the same effect, and is done for the same reason, as Korel and Laski's Identity Relation (IR) [63]). Finally, because each of these steps may have introduced new ops into the slices in ways that require recomputing information in previous steps, we repeat the process until a fixed point is reached.

3.5.3.1 Correctness

We argue the correctness of our trace merging algorithm in two parts. First, the algorithm certainly reaches a fixed point and terminates within a finite number of iterations. The traces used as input are finite in number and length. This means that the number of micro-operations initially marked as "in a slice" by the dynamic slicing algorithm of Korel (which has, itself, been shown to complete in a finite number of steps) is also finite. Thus, the number of micro-operations as-yet unmarked is also finite. With each iteration of the algorithm, we either add no new ops to a slice, in which case we have reached a fixed point and terminate, or we add some finite number of ops to some slices and loop. Thus, the algorithm will, at worst, add one op to each slice with each iteration, ending with every op in every trace marked, but even this will happen in a finite number of iterations of the algorithm.

Second, to see that our algorithm correctly merges multiple traces, consider combining traces T and T', each of which (when we run our algorithm on it individually) produces programs P_T and $P_{T'}$, which each have the correct output given their respective input (we can assume this because in this base case our algorithm is equivalent to that of Korel and Laski [63]). By examining Algorithm 1, when we produce a program P from the combination of T and T', we can see that there are only two cases where P will differ from P_T or $P_{T'}$:

Case 1 We include a branch statement that was not included in T or T' by itself. But by including this branch, we include the ops necessary to decide which way to go at that branch. So when P is run with the inputs provided to T, the branch will evaluate as it did in T, and the successor will be the same as for P_T. The same argument applies when P is

run with the inputs for T'.

Case 2 We include an op (and the ops necessary to compute the dependencies of that op) that was in T and not T' (or vice versa). Assume without loss of generality that the op was in T and not T'. Then the ops added to T' do not affect the output value along the path taken by T' (or they would have been added by the dynamic slice that initialized $slice(T')$). So when P is run with the inputs for T', it will produce the same output as $P_{T'}$.

This argument extends by induction to show that given an arbitrary number of traces $\{T_1, \ldots, T_n\}$, if the corresponding programs $\{P_1, \ldots, P_n\}$ are correct with respect to their inputs, the merged program will be correct with respect to all their inputs. We have also validated the correctness of our algorithm experimentally, as we describe in Section 3.6.2.

3.5.3.2 Fortuitous Coverage Enhancement

We have demonstrated that a merged program P can surely execute correctly on guest states from one of its constituent traces. However, aside from this additive effect, combining traces can also allow the program to generalize better to guest states not seen during training. To see why this is the case, consider Figure 15, which depicts two program paths that are merged by our algorithm. In addition to the two paths covered by each individual trace ($A \to C \to D \to E \to G$ and $A \to B \to D \to F \to G$), the merged program also covers paths $A \to C \to D \to F \to G$ and $A \to B \to D \to E \to G$). Depending on the relationship between the branch conditions at A and D, these paths may be infeasible, but in the best case, the merged program will be able to generalize to guest states that differ from the training states along these paths.

In our experience with Virtuoso, we have noticed several instances of this "fortuitous" coverage improvement. For example, while testing the **pslist** program for Windows (see Section 3.6), we generated two versions of the program from two different training runs. Out of three system states in our testing corpus, we observed the following behavior:

- The first program worked correctly on states 1 and 2, but not state 3.
- The second program worked correctly only on state 1.

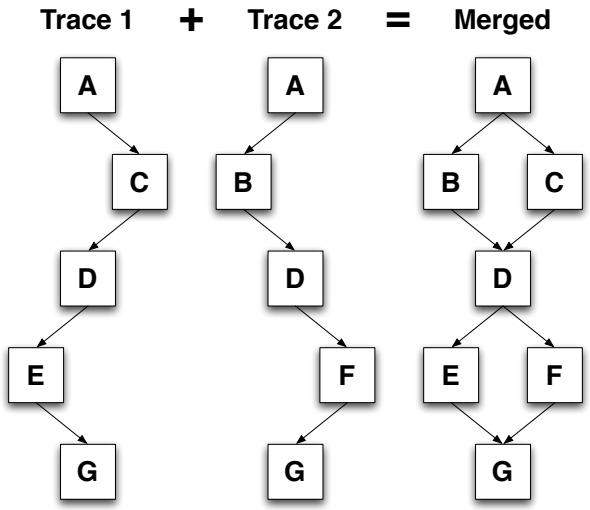


Figure 15: An example of a non-additive coverage improvement from combining two traces. Rather than the expected two paths covered, we in fact cover four program paths.

- The merged program, generated from both training runs, worked correctly on all three system states.

While this is an isolated example, it suggests that such non-additive coverage improvements can boost the reliability of the generated introspection programs. We hope to make this notion more rigorous in future work, and explore its potential for producing more reliable introspection programs in conjunction with static analysis.

3.5.4 Translation and Runtime Environment

Our Instruction Translator (shown below the Trace Analyzer in Figure 9) takes the sliced micro-op traces and creates an equivalent Python program that can be run outside the VM. Currently, the code produced takes the form of a plugin for Volatility [124], a framework for volatile memory analysis written in Python. Volatility was chosen because it already provides certain facilities needed in our runtime environment, such as x86 virtual address translation, and the ability to analyze the memory of a virtual machine running under Xen [9] (through the third-party extension PyXa, included in the XenAccess library [94]). Moreover, it features an API that makes it easy to layer new functionality on top of existing address space objects, which we use to implement copy-on-write semantics for the guest memory (see below).

The translation of QEMU micro-instructions to host code is fairly direct. The translation works one basic block at a time, starting with the output of Algorithm 1. Each individual op is mapped to a simple Python statement. At the end of the block, if there is only one successor, an unconditional jump to that successor is produced. Otherwise, the appropriate conditional or dynamic jump is output.

The runtime environment itself, shown on the right in Figure 9 is implemented as a plugin for Volatility. The introspection tool is output by the trace analyzer as a dictionary of blocks of Python code, keyed by the EIP of the original x86 code. The plugin performs some initial setup, and then loads and executes the block marked "START". When the block finishes, it sets a variable named `label` that determines the successor (this variable can be set conditionally, unconditionally, or dynamically, to implement the three successor cases outlined above). This process continues until the successor returned is "EXIT". At the end, the contents of the output buffer are displayed to the user. The details of the runtime environment, including input and output handling, are presented below.

Registers are modeled as variables in the generated code; thus, to implement copy-on-write behavior for registers, we simply initialize the values of the register variables before the generated code. Any references to a register that occur before the generated code assigns to it will therefore retrieve the value from the guest VM, as desired. Assignments to registers, by contrast, will simply overwrite the Python variable, and will not affect the running VM.

Access to memory is handled by translating load and store operations into `read` and `write` operations on a copy-on-write address space object in Volatility. The COW space is initialized by passing it the currently active virtual address space for the VM (on x86 guests, this boils down to using the current value of the guest's `CR3` register to map virtual addresses to physical). All reads and writes are then done on this space, which implements copy-on-write as described in Section 3.4.3.

The runtime environment also provides a host-side memory allocator for use with the malloc summaries described in Section 3.5.2. Before the program is run, the runtime environment scans the guest page tables for an unused virtual address range. It then adds page table entries (as always, making changes to the copy-on-write space—the guest's state

is not modified) to create a heap that can be addressed as though it were actually inside the guest. To prevent conflicts, the physical pages are reserved by choosing a range above the amount of physical memory available to the guest.[4] Reads and writes to that physical range are redirected to the host memory area.

Inputs to the program are represented by an array named `inputs`. When the Instruction Translator comes across a micro-op that is marked as reading input data, it translates it into a simple assignment that takes the value from the input array. For example, if a micro-op such as `LDL T0, A0` is marked as needing the first input parameter to the program, the code produced will be `T0 = inputs[0]`. The `inputs` array is populated at runtime via a command line option to the Volatility plugin.

Finally, output-producing operations are translated into two separate Python statements. The first performs the write to memory normally, while the second writes the data into a special output memory space and tags it with a label that identifies which output buffer it was associated with. Once the plugin has finished executing the translated code, it dumps the contents of the output memory space. The user can also optionally pass in a function to interpret the output (e.g., by converting it from little-endian Unicode to a string, or to interpret a 64-bit integer as a human-readable time).

3.6 Evaluation

In this section, we evaluate Virtuoso using a number of different criteria: generality, reliability, security, and performance. To evaluate the generality of Virtuoso, we look at the diversity of operating systems and the different types of programs for which introspections can be generated. Next we discuss the *reliability* of the resulting introspection programs—that is, their ability to function correctly when examining a system at runtime. Finally, we examine the resilience to attack of programs output by Virtuoso, and we discuss their runtime performance.

[4] Although this means that in a 32-bit environment we cannot support guests with 4 gigabytes of RAM, we note that this is only a peculiarity of our implementation; see Section 3.7 for details of how we plan to remove this limitation. Also, the move to 64-bit architectures will provide ample physical address space for Virtuoso to use if needed.

3.6.1 Generality

Because Virtuoso uses very little domain knowledge about the target OS, it is easy to generate new introspection programs for operating systems that run on the x86 architecture. To demonstrate this capability, we created six training programs for three different OSes. These programs were chosen because they represent common actions that might be needed for passive and active security monitors: a security system may need to list drivers to audit the integrity of kernel code, or enumerate processes in order to scan them for malicious code, and so on. Some of these functions are also implemented for Windows targets in tools such as Volatility [124]; however, our own tools required no reverse engineering and were generated automatically.

The six programs performed the same function on all three operating systems:

- **getpid** Gets the process ID of the currently running process.
- **gettime** Retrieves the current system time.
- **pslist** Computes a list of the PIDs of all currently running processes.
- **lsmod** Retrieves a list of the base addresses of all loaded kernel modules.[5]
- **getpsfile** Retrieves the name of the executable associated with a given PID.
- **getdrvfile** Retrieves the name of a loaded kernel module, given its base address.

The three operating systems used were Windows XP SP2 (kernel version 5.1.2600.2180), Ubuntu Linux 8.10 (kernel version 2.6.27-11), and Haiku R1 Alpha 2 [46]. Windows and Linux were chosen because they are both in common use,[6] and thus represent practical, real-world scenarios. Haiku, a relatively obscure clone of BeOS, was chosen for two reasons: (1) to our knowledge, we are the first to do virtual machine introspection or memory analysis on a Haiku target, making it a compelling example of Virtuoso's ability to deal with diverse

[5] In the Linux version, this reads the contents of `/proc/modules`, which contains both the names and base addresses of kernel modules, so `getdrvfile` is not needed on Linux.

[6] Although use of OS X is also widespread, its design isolates the kernel in a separate address space. Due to limitations in our current implementation discussed in Section 3.7, we were unable to test OS X introspection.

OS	Program	Original	Post	Final
Windows	getpid	3549	106	12
	gettime	7715	1081	233
	pslist	302082	230366	75141
	lsmod	195488	157440	84973
	getpsfile	49588	23865	10074
	getdrvfile	194765	157696	92109
Linux	getpid	133047	4055	1706
	gettime	75074	3882	1592
	pslist	6107214	2265667	1095963
	lsmod	1936439	852299	414670
	getpsfile	14752561	6323249	2913064
Haiku	getpid	18242	3985	1719
	gettime	9982	585	237
	pslist	362127	290078	160830
	lsmod	850363	702277	423438
	getpsfile	249663	152896	78107
	getdrvfile	522299	399175	234527

Figure 16: Results of testing a total of 17 programs across three different operating systems. "Original" refers to the size of the trace before any processing, "Post" is the size after interrupt filtering and malloc replacement, and "Final" gives the size of the program after slicing; all sizes given are in number of IR ops. The numbers given refer to processing of a single dynamic trace.

operating systems; and (2) none of the authors had any prior knowledge of the internals of the Haiku kernel, so there was no way to "cheat" by incorporating additional domain knowledge.

The results of our testing are shown in Figure 16. To verify that each program worked correctly, we ran it on a sample memory image taken at a different time from when the trace was captured, and then verified the output by hand (with help from existing in-guest tools). The numbers shown are for programs generated from a single trace.

With one exception, the generated programs all worked correctly—`gettime` for Haiku failed to obtain the correct system time. Upon inspection, we determined that this was due to the way Haiku keeps time: rather than continuously updating some global location, Haiku stores the boot time and then calculates the current time by calling `rdtsc` to read the Time Stamp Counter (TSC), which contains the number of cycles executed since boot. Because our runtime environment does not have access to the TSC when operating on a memory image, the time returned by `gettime` is incorrect. However, we manually verified

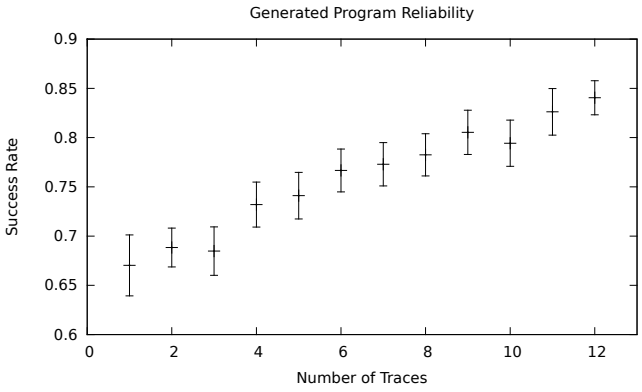


Figure 17: Results of cross-validation for the `pslist` program on Windows. The error bars indicate the standard error of each sample.

that when provided with the correct TSC value, the generated program functions correctly.

Figure 16 also shows the effect of preprocessing and slicing on the size of the generated program. One clear point emerges from examining these figures. Preprocessing removed between 17% and 97% of the initial traces (it removed 55% on average), and slicing reduced the number of remaining instructions in the trace by 40%–89% (56% on average). Because the resulting programs do, in fact, work, we can conclude that a large portion of the computation seen in the initial traces is irrelevant to the final output. Also, we can observe that were we to run all the code seen in the trace, the time required to run the introspection programs would dramatically increase.

3.6.2 Reliability

Because our introspection programs are based on dynamic analysis of the training programs, they do not, in general, contain all the code needed to account for every eventuality the introspection program may encounter. This limitation is inherent to the use of dynamic methods and is a well-known problem in program testing. In this section, we describe the results of our experiments on the reliability of the generated introspection programs, techniques for improving their coverage, and the motivation behind our use of dynamic analysis.

Testing To empirically test the reliability of programs generated using Virtuoso, we examined how the reliability of a single program (`pslist`, described in Section 3.6.1) was affected by training. We generated 24 traces of `pslist` running under Windows XP, taking a trace once every five minutes, starting just after the desktop appeared. During this time, we did the following actions:

- Opened a connection to a remote machine with PuTTY.
- Browsed web pages with Google Chrome and Internet Explorer.
- Played a game of Minesweeper.
- Closed a non-responsive instance of Internet Explorer, and sent a crash report.
- Played a game of Solitaire.

No special effort was made to induce a wide variety of system states such as low-memory conditions. Along with each trace, we also captured a snapshot of the memory and CPU state.

We then used cross-validation to estimate the reliability of the program as a whole: for each k in the range $\{1 \ldots 12\}$ we took 50 random subsets of size k from the set of all traces captured,[7] used these traces to generate an introspection program, and then tested the reliability of the program on the $(24-k)$ images corresponding to the remaining traces. We judged that the generated program had executed correctly if it produced the same output as the training program for that image. Figure 17 shows the results of this testing. Each point represents the mean reliability of programs generated from k traces.

We note that reliability appears to increase linearly with the number of traces used; since the reliability is bounded above by 1, it most likely approaches an asymptote as more traces are added and the number of unexplored paths in the program grows smaller. We also ran the generated programs on their training images, and, as expected, achieved a success rate of 100%, validating the correctness of our trace merging algorithm.

[7] For $k = 1$ there are only 24 subsets, so our number of subsets in this case is 24.

In examining the data more closely, however, we observed that traces taken earlier in time (closer to system boot) were more reliable. To test this effect, we created an additional test data set of 24 images, again captured once every five minutes starting shortly after system boot. We then generated the `pslist` program from the first trace captured during our cross-validation test and no others. To our surprise, we found that this program had 100% reliability on our new test data set.

By inspecting the source code to the Windows Research Kernel [84], we determined that the Windows operating system stores the image name of the process *lazily*, waiting to generate the attribute until the first time some program requests it. So the first trace includes all the code needed to generate the process name, whereas later traces simply read the already-initialized data. When programs produced from these latter traces encounter a case where the process name is not yet initialized, they will lack the code necessary to do so. Thus, this caching effect may cause the cross-validation numbers given above to overestimate the difficulty of building reliable programs; in this case, the natural testing strategy of booting the system and taking several traces would have yielded a reliable program.

Improving Coverage In any case where testing finds that the generated program fails to function correctly, we can take a new trace based on the failing snapshot that should work correctly in that case. Although we will not be able to run the program precisely from the point where the failure occurred, in practice we have found that failing states tend to persist long enough that we can take a new trace that covers the failed snapshot. This allows us to iteratively improve coverage based on gaps found during training, until we reach a point where no further gaps are found. In future work, we hope to explore how techniques from the software testing community can help make this approach more rigorous. For example, we may be able to perform static analysis of the OS code to reveal missing branches, and then apply symbolic or concolic execution to determine what system state is necessary to go down that branch and improve coverage.

Static vs. Dynamic Analysis A natural question is why, given the coverage issues associated with dynamic analysis, our approach does not make use of static analysis. Although we considered static analysis, a number of difficulties make it an unattractive choice in this context. First, static analysis relies on the ability to reliably identify all the code associated with a piece of functionality. In the general case, accurate disassembly of x86 code is undecidable [78], and even on non-malicious code, it can be quite difficult. Second, our slicing approach would be much less accurate if performed statically, particularly in the presence of dynamic jumps (which are common in kernel code, which makes extensive use of function pointer-based indirection) and dynamically allocated memory. Finally, static analysis requires much more domain knowledge about the target operating system: at minimum, the executable file format and the details of the loader must be known just to find the code to analyze.

To demonstrate the reasons why static analysis is inappropriate in this case, it is useful to look at a recent system, BCR (Binary Code Reutilization) [18], which also attempts to extract portions of a binary program for later re-execution, but does so using static analysis. First, we note that BCR does not use a purely static approach: it relies on dynamic analysis in its disassembly phase to resolve the targets of dynamic jumps. The authors do not clearly indicate how their system deals with any coverage issues that may arise from this use of dynamic analysis. Second, BCR employs significant amounts of domain knowledge to perform its code analysis. The executable file format, system API calls, and the mechanism by which imported functions are resolved must all be known in order to extract assembly functions. This reliance on domain knowledge both restricts the generality of BCR and limits the kinds of code it can extract—for example, BCR is unable to extract functions that make system calls directly. Virtuoso, by contrast, relies on no such domain knowledge and works with any code constructs supported by the x86 architecture.

Even if it were possible in our case to statically find every code path that could be taken by the program, it may not be desirable. For example, consider the case of a program written for Linux that retrieves sensitive information from the `/proc` filesystem. At some point in the execution of this program, the kernel will likely perform a check along the lines

of `if (uid == 0)`. Although this check is a valid part of the overall program, it does not actually affect the *value* computed by the program. If we use dynamic analysis and collect traces that succeeded, we will only ever see the branch where this condition evaluated true, and the check will not be incorporated into the resulting program. Although this is, strictly speaking, incorrect, it has the side effect of *generalizing* the program: whereas the original code could only be executed in a context where the current user was root, the generated introspection utility will be able to compute the correct result from the context of any user. Additional research is required to determine if these "undesirable" branches can be automatically eliminated using a static approach.

3.6.3 Security

The main motivation in the design of Virtuoso is to enable the rapid creation of secure introspection-based programs. Were security and unobtrusiveness of no concern, it would be sufficient to deploy an in-guest agent that queried existing OS APIs and simply trust that they had not been compromised. However, as security *is* a concern, we must test that our system does, in fact, provide the hoped-for isolation from malicious changes to the guest operating system.

To demonstrate that the generated programs are immune to malicious changes in kernel code, we generated our Windows process lister (`pslist`, described in Section 3.6.1) on a clean Windows XP system. Next, we obtained the Hacker Defender [36] rootkit, which (among many other stealth techniques) hides processes by injecting into all processes on the system and hooking API calls using inline code modification. We configured it to hide any process named `rcmd.exe`. We then renamed a copy of Notepad to `rcmd.exe`, and infected the system. After infection, we checked that `rcmd.exe` was no longer visible from the Task Manager. We then ran our generated introspection program, and found that it successfully listed both our hidden process and the rootkit's own userland component (`hxdef100.exe`).

Beyond verifying that our program is immune to existing rootkits that alter kernel code, we also need to consider the possibility that someone might actively attempt to evade our introspections. First, because Virtuoso attempts to faithfully replicate the functioning

of actual system APIs, vulnerabilities that allow attackers to evade the standard APIs will be reflected in the generated introspection programs. Data-only attacks (i.e. Direct Kernel Object Manipulation, or DKOM) also poses a challenge to our system, because in this case even uncompromised OS APIs will report incorrect results. For example, the standard Windows process listing API can be evaded by rootkits that directly manipulate the process data structure to unlink a malicious process, hiding it from the `EnumProcesses` API. Careful combinations of existing introspections can still thwart this attack, however: by calling **getpid** every time the `CR3` register changes and comparing this list against the one returned by **pslist**, hidden processes can still be detected. Further research is necessary to see if such defenses can be found for other kinds of data-only attack.

Second, an attacker may attempt to take advantage of the fact that Virtuoso uses dynamic analysis to cause it to malfunction. Specifically, he may attempt to set the *free variables* used by the program in such a way as to cause some conditional branch that does not have full coverage to be exercised. The opportunity for evasion afforded by this is minimal: if an attacker does find a way to reliably manipulate the system into a state that causes an introspection program to fail, the fix can be generated quickly, as the attack could be used for a new training run that would update the existing introspection program. Extra care must be exercised in this case, however: if the attack also alters code that the introspection depends upon, updating the introspection program could cause malicious code to be incorporated into the generated program. Kernel code integrity checks (i.e., verifying that the code seen in training matches that found in an uncompromised version of the OS) would help in this scenario, but can be difficult to implement in practice, and would require more domain knowledge than we currently assume.

Additionally, we note that our dynamic slicing method allows us to quantify precisely the set of global kernel data that a given introspection depends on (note that data from userland is read from the training data, and is not available to the attacker for manipulation). Even within this set of kernel data, only a subset may be susceptible to attacker manipulation: as Dolan-Gavitt et al. [33] describe, arbitrary modification of global kernel data can cause system instability. Ultimately, this problem is best resolved by improving training, and we

hope to explore automated means of improving coverage in future work.

The general problem of defending against malware that is VMI-aware, and has the ability to tamper with kernel data structure layouts and algorithms, remains open. Such malware, which would affect most current VMI solutions, has been discussed in other work [5], and defending against it is a difficult problem. Although we consider this problem out of scope for Virtuoso, we hope to explore more general defenses in future work.

Finally, if a vulnerability is found in the APIs that support the introspection, an attacker may attempt to compromise the security of the monitor by causing it to execute malicious code. Because the runtime environment has no facility for translating new code, it is effectively a Harvard architecture, so standard code injection techniques are not possible against programs generated by Virtuoso. However, in recent years a new technique, *return-oriented programming* [106], has demonstrated that malicious computation can be performed by chaining together snippets of code linked by dynamic jumps (e.g., returns), and subsequent work has shown that this allows for exploitation of Harvard architecture devices [20]. Despite these developments, we believe that the relatively small size of our programs (the largest in our test set has only 4030 basic blocks, and only 140 of these contain a dynamic jump), combined with the fact that the runtime environment executes code at the granularity of a basic block (i.e., it is not possible to execute only part of a block), will make it impossible to construct the necessary set of gadgets. More work, however, is necessary to prove this intuition definitively.

3.6.4 Performance

Performance results for the programs described in Section 3.6.1 are shown in Figure 18. The tests were carried out on a Intel® Core™ 2 Quad 2.4 GHz CPU machine with 4 gigabytes of RAM running Debian/GNU Linux unstable (kernel 2.6.32-amd64). Each test was run 100 times, and the results were averaged.

Of the three operating systems shown, the results for Linux stand out as being particularly slow. We investigated this result further and found that the overhead was caused by the interface Linux uses to retrieve system information. In contrast to Haiku and Windows,

OS	Program	Time (ms)
Windows	getpid	0.2
	gettime	1.5
	pslist	450.2
	lsmod	698.1
	getpsfile	59.8
	getdrvfile	751.9
Linux	getpid	9.6
	gettime	9.6
	pslist	6394.1
	lsmod	2437.0
	getpsfile	20723.9
Haiku	getpid	33.7
	gettime	1.8
	pslist	712.1
	lsmod	3351.7
	getpsfile	479.6
	getdrvfile	1901.0

Figure 18: Runtime performance of generated programs. Times given are in milliseconds, and are averaged over 100 runs.

which have specific system calls to inspect the state of the system, Linux exposes these details through the `/proc` filesystem. This means that the code that implements tools such as `pslist` needs to open files, list directories, and so on, all of which results in much more code being executed than on Windows and Haiku.

Although the times shown are not currently fast enough to enable online monitoring, we stress that our current implementation is unoptimized and translates the x86 code to Python, which it then runs in an environment that is also written in Python. One performance improvement would be to port the runtime to C and generate x86 binary code that can be run on the host (after transforming potentially dangerous instructions into safe equivalents, redirecting memory loads and stores to the guest VM, and so on). We expect that this would provide performance at least on par with QEMU, which uses similar techniques in its whole-system emulation. It is also worth noting that even unoptimized, the programs generated by Virtuoso are fast enough for use in forensic analysis of memory dumps.

3.7 Limitations and Future Work

Although Virtuoso currently supports many useful introspections on a variety of operating systems, there are still a number of areas in which it could be extended. In this section, we describe the current limitations of Virtuoso and how we might deal with these cases.

Multiple address space support: Perhaps the most significant limitation of Virtuoso is its inability to correctly handle traces that span multiple virtual address spaces (that is, traces in which another process than the training program appears). In the simplest case, the other processes do no work related to the introspection at hand, but they are difficult to filter out because the trace will also contain portions of scheduler code to switch to the other process, and identifying and removing this code automatically is difficult. Virtuoso currently sidesteps this issue by running its training programs with high priority (e.g., using `start /realtime` on Windows and `chrt` on Linux).

More difficult, however, is the problem of analyzing and extracting programs that make use of interprocess communication (IPC). Solving this problem is particularly important for supporting introspection of microkernel architectures, where many tasks are performed by collections of cooperating processes. The major challenge here is that although we may be able to extract the relevant code running in each process, there will be data from other processes that must be read from the guest operating system at runtime. The generated program, then, will need to have some way of finding the appropriate cooperating process at runtime in order to read this data; this is likely to require significant domain knowledge. More research is required to determine the extent to which this can be automated.

Self-modifying code: As we mention in Section 3.6.3, the runtime environment of Virtuoso is effectively a Harvard architecture machine, with no facility for loading new code. This means that we do not support self-modifying code. Although this has not caused problems with any of our introspections (operating systems rarely make use of self-modifying code), such support is necessary for working with malicious code, an area we hope to explore in the future.

Relocation and ASLR: Currently, Virtuoso assumes that the modules containing the data it needs have not been relocated. This assumption may not hold in general, however:

modules may be loaded at different base addresses for a variety of reasons, including security (i.e., Address Space Layout Randomization, or ASLR).[8] If this occurs, data read from those modules will be found at a different address, and the introspection program may not be able to locate it correctly. Although this problem can be resolved by incorporating more domain knowledge (such as the mechanics of the executable loader), doing so would make supporting new operating systems more difficult. One possible solution to this problem is to attempt to *relocate* portions of the generated program at runtime by scanning the memory of the guest operating system for the relevant code and then applying the appropriate offset for accesses to memory.

Aside from improving the analysis capabilities of Virtuoso, we also hope to examine its use in areas outside the realm of introspection. For example, by tracing calls made by programs such as `ls`, we might be able to produce programs that can understand the format of undocumented filesystems without relying on native filesystem drivers. Such analysis capabilities could aid significantly in cross-platform interoperability efforts.

Finally, there are some minor implementation details that could be improved. In particular, we could do away with needing to know the details of malloc for specific operating systems by adding support for x86 hardware exceptions and extracting the page fault handler. As we already detect page faults in our runtime environment (they currently raise an exception), we would just need to extract the page fault handler code and cause memory exceptions to trigger this code (similar to the way the TPR is currently handled—see Section 3.5.2 for details). This would eliminate the only piece of OS-specific knowledge used by Virtuoso, allowing it to generalize to any x86-based operating system.

3.8 Conclusion

We have presented Virtuoso, a system for automatically generating introspection tools that can retrieve semantically meaningful information based on low-level data sources. By applying a novel whole-system executable dynamic slicing technique, Virtuoso turns a task which

[8]The alert reader will note that the version of Linux used enables ASLR by default. However, this did not affect our introspections, as the only libraries randomized are in user space, and we take userland data references from training.

once took hours or weeks of reverse engineering by an expert into one that requires only a small amount of effort by a programmer of modest skill and a few minutes of computation time—and in doing so, helps ensure that introspection programs exactly model the behavior of the operating system. Moreover, its analysis capabilities are operating system-agnostic, removing the need for developers to constantly play catch-up as OS vendors release new versions of their products. These contributions help narrow the semantic gap, and should remove a significant roadblock in the areas of forensic analysis, virtualization-based security and low-artifact malware analysis.

CHAPTER IV

TAPPAN ZEE (NORTH) BRIDGE: MINING MEMORY ACCESSES FOR INTROSPECTION

4.1 Motivation

Many security applications have a need to inspect the internal workings of software. Host-based intrusion detection systems, malware analyses, and digital forensics all depend to some degree on being able to obtain information about software that is by design undocumented and hidden from public view. Thus, to operate correctly, security software is typically built on *reverse engineering*, the art and practice of elucidating the undocumented principles on which software is built.

Unfortunately, reverse engineering is expensive, time consuming, and requires a high degree of expertise. The problem is exacerbated by the fact that, to protect against tampering, security applications are often hosted in environments separated from the target being inspected, such as a separate virtual machine. Because of this, their visibility into the target is often limited to low-level features such as memory and CPU state, and any higher-level information must be reconstructed based on reverse engineered knowledge.

This problem, which we will refer to as the *introspection problem*, has been approached by a number of recent research efforts such as Virtuoso [32] and VMST [40]. Existing systems, however, have a number of limitations. First, they focus on retrieving kernel-level information. However, a great deal of security-relevant information exists only at user-level, such as URLs being visited by the browser, instant messages and emails sent by desktop clients, and system and application log messages. Second, they require that the desired information be accessible through some public interface (a public API in the case of Virtuoso, and a userland program or kernel module in the case of VMST). This means that some security-relevant information may be inaccessible to such tools. Finally, Payne et al. [95] argue that many security applications need some form of *active monitoring*; that

is, they need to be notified when certain system events occur. Current solutions to the introspection problem provide no way of locating places in the system where it would be useful to interpose.

In this chapter, we attempt to address the limitations of past solutions by examining a rich source of information about system and application activity: memory accesses observed at runtime. Our key insight is that a memory accesses made at different points in a program can be treated as streams of related information. For example, when visiting a URL, a web browser must write to memory the URL that is being visited, and it will generally do so at the same point in the program. By intercepting memory accesses made at this program point we can observe all URLs visited. These program points, which we call *tap points*, provide a natural place to interpose to extract security-relevant information, and could be integrated into an active monitoring system such as Lares [95].

There are several challenges that must be overcome to make use of tap points. The first is the sheer amount of data that must be sifted through. In ten minutes' worth of execution on a Windows 7 system, for example, we observed a total of 18.9 *million* unique tap points which read and wrote a total of 32.8 gigabytes of data. To overcome this challenge, we make use of techniques from information retrieval and machine learning, described in Section 4.4, to quickly zero in on the tap points that read or write information relevant to introspection.

Second, simply setting up an environment in which one can observe every memory access made by the whole system (OS and applications) poses a challenge. Whole-system emulators such as QEMU [12] provide the necessary basis for such instrumentation, but intercepting and analyzing every memory access online is not practical: the resulting system is so slow that network connections time out and the guest OS may think that programs have become unresponsive. To solve this problem, we add *record and replay* to QEMU, which allows executions to be recorded with low overhead. Our heavyweight analyses are then run on the replayed execution to analyze every memory access made *without* perturbing the system under inspection. We describe our system, Tappan Zee Bridge (TZB),[1] in detail in

[1]So named because the northbridge on Intel architectures traditionally carried data between the CPU and RAM.

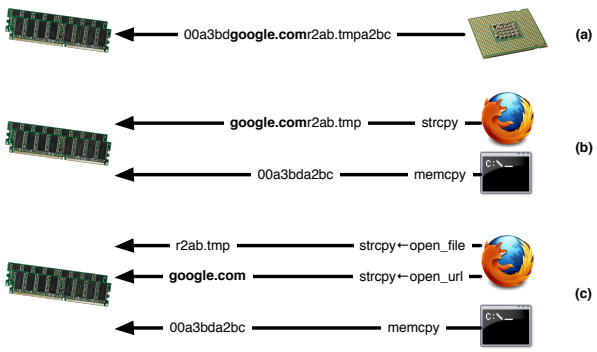


Figure 19: Three different ways of defining a tap point: (a) as a single stream of information from the CPU to RAM ; (b) split up according to program and location within program ; (c) split up according to program, location within program, and calling context.

Section 4.5.

Finally, previous systems have required significant effort to support new architectures. This problem has become more pressing in recent years, as ARM-based devices such as smartphones have exploded in popularity. Because TZB looks at memory accesses, rather than inspecting binary code, it naturally supports a wide variety of architectures with minimal effort. To demonstrate this, our evaluation includes the ARM architecture in addition to x86, and the techniques we describe easily generalize to other architectures.

4.2 Defining Tap Points

At the heart of our approach is an abstraction on top of memory accesses made by the CPU, the *tap point*. A tap point is a point in a system at which we wish to capture a series of memory accesses for introspection purposes; however, the exact definition of "a point in a system" will make a great deal of difference in how effective our approach can be.

A naive approach to defining tap points would be to simply group memory accesses by the program counter that made them (e.g., EIP/RIP on x86 and R15 on ARM). This approach fails in two common cases: first, memory accesses made by bulk copy functions, such as `memcpy` and `strcpy`, would all be grouped together, which would commingle data from different parts of the program into the same tap point. In addition, looking only at the program counter would conflate accesses from different programs.

Instead, we define tap points as the triple

$$(caller, program\_counter, address\_space)$$

Including the caller and the address space (the `CR3` register on x86, and the `CP15 c2` register on ARM) separates out memory accesses into streams that should, in general contain the same type of data.[2] Figure 19 shows the effect of choosing various definitions of a tap point when looking for the place where the browser writes the URL entered by the user ("google.com"). At the coarsest granularity (a), one can simply look at all writes from the CPU to RAM; however, the desired information is buried among reams of irrelevant data. Separating out tap points by program and program counter (b) is better, but still combines uses of `strcpy` that contain different information — in this case, a filename and a URL. By including the calling context (c), we can finally obtain a tap point that contains just the desired information.

It is possible that some tap points may require deeper information about the calling context (for example, if an application has its own wrapper around `memcpy`), but in practice we have found that just one level of calling context is usually sufficient. In addition, because TZB uses a whole-system emulator that can watch every call and return, we can obtain the call stack to an arbitrary depth for any tap point. This makes it easy to add extra context for a given tap point, if it is found that doing so separates out the desired information. Examples of tap points that require more than one level of callstack information are given in Sections 4.6.1.2 and 4.6.1.3.

Conversely, one might wonder whether this definition of a tap point may split up data that should logically be kept together. To mitigate this problem case, we introduce the idea of *correlated tap points*: we can run a pass over the recorded execution that notices when two tap points write to adjacent locations in memory in a short period of time (currently 5 memory accesses). The idea is that these tap points may be more usefully considered jointly; for example, a single data structure may have its fields set by successive writes.

[2]Making use of tap points defined this way in the real world is slightly more difficult, since a program's address space will differ and its code may be relocated by ASLR. These complications can be overcome with a minor amount of engineering, however.

These writes would come from different program counters, and hence would be split into different tap points, but it may be more useful to examine the data structure as a whole. By noticing this correlation we can analyze the data from the combined tap point.

4.3 Scope and Assumptions

The goal of Tappan Zee Bridge is to find points at which to interpose for active monitoring. More precisely, our goal is to speed the current entirely manual process by which applications or operating systems are reverse engineered in order to locate tap points for active monitoring. It should be noted that we do not aim to surpass those manual efforts. We have no automatic way, for instance, of knowing for certain if a tap point will fail to output crucial data or, alternately, spew out superfluous information under some future conditions. This is a separate problem to which we see no ready solution. Static analysis of candidate tap points or extensive testing are good stop-gaps, but nothing short of fully understanding enormous binary code bases can really give complete assurance that a tap point won't miss or cause false alarms in the future.

In this section, we explore how our definition of a tap point and our focus on active monitoring shape the scope of our work.

First and most obviously, our focus on memory accesses necessarily limits our scope to information that is read from or written to RAM at some point. Although this is quite broad, there are notable exceptions. For example, TRESOR [89] performs AES encryption without storing the key or encryption states in RAM by making clever use of the x86 debug registers and the AES-NI instruction set. Aside from such special cases, however, this assumption is not particularly limiting.

Second, our goal of finding tap points suitable for active monitoring motivates a design that treats memory accesses at tap points as sources of *streaming* data. Our algorithms, therefore, typically work in a streaming fashion as the system executes, remembering only a fixed amount of state for each tap point. Although this is a natural fit for active monitoring, where events should be reported as soon as possible, it makes handling data whose *spatial* order in memory differs from its *temporal* order as it is accessed more difficult.

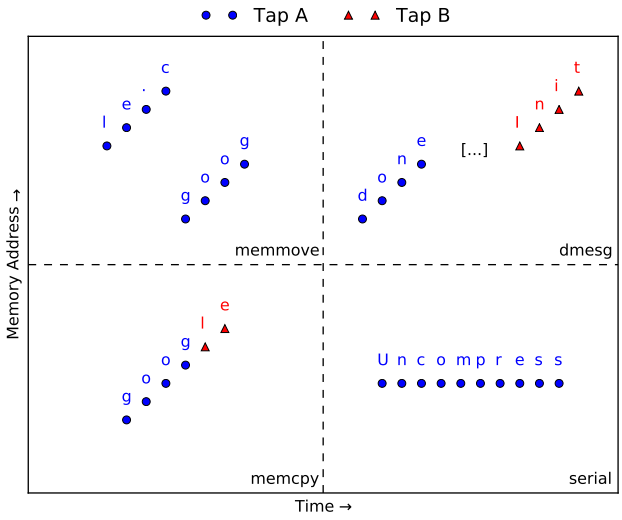


Figure 20: Patterns of memory access that we might wish to monitor using TZB.

Third, the *encoding* of the data sought must be to some extent guessable. For example, to search for a string, one must know what encodings are likely to be used by the system to represent strings. In general this is not a severe limitation, but it does come up; we discuss one such case in Section 4.6.2.2.

Finally, the use of calling context in the definition of a tap point raises the question of how much context is necessary or useful. Our current system uses only the most recent caller, but we have seen both situations where this is not enough and where it is too much. Overall, however, one level of calling context has proved to be a reasonable choice for a wide variety of introspection tasks.

To better illustrate the boundaries of our technique, consider Figure 20, which plots the address of data written by different tap points over time for four patterns of memory access. In the bottom two quadrants, we have cases that are challenging, but currently well-supported by TZB. In the bottom-left, a standard `memcpy` implementation on x86 makes a copy in 4-byte chunks using `rep movsd`, and then does a two-byte `movsw` to get the remainder of the string. Because the access occurs across two different instructions, TZB sees two different tap points. Our tap point correlation mechanism correctly deduces that the accesses are related, however, because they operate on adjacent ranges in a short span of time.

The case shown in the bottom right quadrant would be tricky if we looked only at memory access spatially and not temporally. Here, a utility function writes data out to a serial port by making one-byte writes to a memory-mapped I/O address.[3] Because TZB sees these memory writes in temporal order, ignoring the address, the data is seen normally and the analyses we describe all operate correctly.

The upper quadrants show cases that are currently not handled by TZB. In the upper left, `memmove` copies a buffer in reverse order when the source and destination overlap. Thus, when viewed in temporal order, a copy of a string like "12345678" would be seen by TZB as "56781234". This case is unlikely to be handled by TZB without a significant redesign, as its view of memory accesses is inherently streaming.

Finally, the upper right, which represents the case of `dmesg` on Linux, is an example of the "dilemma of context". Although the function, `do_syslog`, that writes log data to memory is called from multiple places (creating multiple tap points), it writes to the same contiguous buffer. Unlike the `memcpy` case, a significant amount of time may pass before the next function calls `do_syslog`, and so our tap correlation, which only considers memory accesses within a fixed time window, will not notice that the tap points ought to be grouped together. We believe that this case could be overcome with additional engineering work, but this is left to future work.

4.4 Search Strategies

To find useful *tap points* in a system—places from which to extract data for introspection—using Tappan Zee Bridge, one begins by creating a recording that captures the desired OS or application behavior. For example, if the end goal is to be notified each time a user loads a new URL in Firefox, one would create a recording of Firefox visiting several URLs. This recording is made by emulating the OS and application inside of the dynamic analysis platform PANDA (described in more detail in Section 4.5.1), which can capture and record all sources of non-determinism with low overhead, allowing for later deterministic replay. Next, one can run one or more analyses that seek out the desired information among all

[3] Although not reported here, this case is one we actually encountered while experimenting with an embedded firmware.

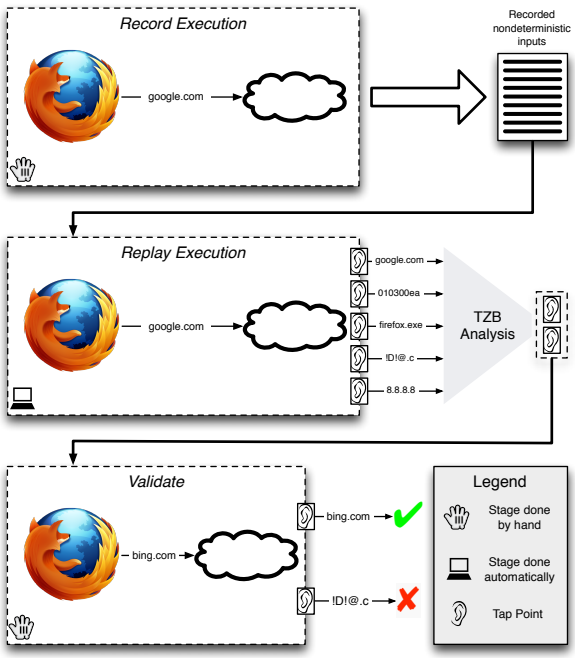


Figure 21: The workflow for using TZB to locate points at which to interpose for active monitoring.

memory accesses seen during the execution. Analyses in TZB take the form of PANDA plugins that are called on each memory access made during a replayed execution and, at the end, write out a report on the tap points analyzed. Finally, the tap points found should be validated to ensure that they do, in fact, provide the desired information. Such assurance can be gained either by examining the data in the tap point in new executions, or by examining the code around the tap point. This workflow is illustrated in Figure 21.

In this section, we describe three different ways of finding tap points grouped according to a standard epistemic classification scheme [101]: searching for "known knowns"—tap points where the content of the desired data is known; searching for "known unknowns"—tap points where the kind of data sought is known, but its precise format is not; and finally "unknown unknowns"—tap points where the type and format of the data sought are not known, and we are instead simply trying to find "interesting" tap points.

4.4.1 Known Knowns

The simplest case is finding data that one knows is likely to be read or written by a tap point, and where the encoding of the data is easily guessed. For example, to find a tap point that can be used to notify the hypervisor whenever a URL is entered in a browser, one can visit a known sequence of URLs, and then monitor all tap points, searching for specific byte sequences that make up those URLs. The same holds for other data whose representation when written to memory is predictable: filenames, window titles, registry key names, and so on. For this kind of data, simple string searching is usually sufficient to zero in on the few tap points that handle the data of interest, and in our experience it is one of the most effective techniques for finding useful tap points.

4.4.2 Known Unknowns

A second tap point application involves finding tap points for things about which we have limited knowledge. We can easily assemble corpora of exemplars to represent a semantic class: English prose, kernel messages, or mail headers. These examples need not come from tap points but can easily be collected directly from interacting with the operating system itself. From such a corpus, we can readily build a statistical model, with which we can build a distance measure for scoring and ranking tap points by how close their contents are to the model.

In addition to such statistical methods, we can also search using an oracle. This is the case, for example, with tap points that write encryption keys. Although the exact key may not be known in advance, we can check whether a given byte string is a valid decryption key by trying to decrypt our sample data.

4.4.3 Unknown Unknowns

The final strategy for finding useful tap points is also the least focused. If there is no specific introspection quantity sought, one might instead wish to find interesting tap points, for some suitable definition of "interesting." To support this scenario, TZB offers a form of unsupervised learning—clustering—to group together tap points that handle similar data.

The idea is that one can then examine exemplars from each cluster, rather than being forced to look through a large number of tap points. Thus, our use of clustering functions as a form of *data triage*.

4.5 Implementation

In this section, we describe both the dynamic analysis platform employed to build TZB, but also TZB-specific algorithmic and data-structure solutions.

4.5.1 PANDA

TZB makes extensive use of the Platform for Architecture-Neutral Dynamic Analysis (PANDA), which was developed by the authors in collaboration with Northeastern University.

PANDA is based upon version 1.0.1 of the QEMU machine emulator [12]. QEMU is an excellent and common choice for whole-system dynamic analysis for two main reasons. First, performance is good (about 5x slowdown over native). Second, every basic block of guest code is disassembled by the host in order to emulate, which means that there are opportunities to interpose analyses at the basic block or even instruction level, if desired. QEMU lowers instructions to an intermediate language (IL) in order to employ a single back-end code generator, the Tiny Code Generator (TCG). This IL means dynamic analyses can potentially be written once and re-used for all 14 architectures supported by QEMU. Further, this version of QEMU is capable of booting and running modern operating systems such as Windows 7 (earlier versions of QEMU such as 0.9.1 cannot).

There are three main aspects to PANDA that make it very convenient for building dynamic analyses. First, PANDA provides a plug-in architecture that readily permits writing guest analyses in C and C++. Plug-in code is executed from a number of standard callback locations: before and after basic blocks, memory read and writes, etc. This is not unlike the schemes employed in other whole-system dynamic analysis platforms such as BitBlaze [112] and S2E [22]. In addition, plugins can export functionality that can then be used in other plugins, allowing complex behavior to be built up from simple components. From a software engineering perspective, PANDA's plugin architecture allows the various analyses supported by TZB to be cleanly separated from the main emulator, which makes

for a much more comprehensible and maintainable codebase.

The second aspect of PANDA that makes it an excellent dynamic analysis platform is nondeterministic record and replay (RR). In our formulation of RR, we begin a recording by invoking QEMU's built-in snapshot capability. Subsequently, we record all inputs to the CPU, including `in`s, interrupts, and DMA. Recording imposes a small overhead (10-20%) but not enough to perturb execution. During replay, we revert to a snapshot and proceed to pull CPU inputs from a log when required. Unlike many other RR schemes, we do not record and replay device inputs, which means we cannot "go live" at any point during replay. But we can perform repeated replays of an entire operating system under arbitrary instrumentation load without worrying about this perturbing application or operating system operation. This capability is vital to TZB: without record and replay, the heavyweight analyses we perform would make the system unusably slow.

The final aspect of PANDA worth mentioning is its integration of LLVM. QEMU lowers basic blocks of guest code to its own IL, which PANDA can, additionally, re-render as basic blocks of LLVM code via a module extracted from S2E. We omit further discussion of this capability as it is not used by TZB.

4.5.2 Callstack Monitoring

As explained in Section 4.2, tap points need information about the calling context. Keeping track of this information requires some knowledge about the CPU architecture on which the OS is running, and so we decided to encapsulate this task into a single plugin. TZB's other analyses can then query the current call stack to arbitrary depth by invoking `get_callers` and not worry about the details described in this section.

To track call stack information, the `callstack` plugin examines each basic block as it is translated, looking for an (architecture-specific) call instruction (currently, we look for `call` on x86 and `bl` and `mov lr, pc` on ARM). If the block includes a call instruction, then we push the return address onto a shadow stack after each time that block executes.

Detecting the return from a function does not require any architecture-specific code. Before the execution of every basic block, we check whether the address we are about to

execute is at the top of the stack; if so, we pop it. We only need to check the starting address of the basic block, because by definition a return terminates a basic block, so the return address will always fall at the beginning of a block.

We note that these techniques may fail if traditional call-return semantics are violated. For example, if a program emulated calls and returns by manually pushing the return address and using a direct jump, it would not be detected as a call. However, for non-malicious compiler-generated code, we have found that the algorithm described here works well.

4.5.3 Fixed String Searching

Searching for fixed strings is one of the most effective tools for finding useful tap points. Because we have to sift through many gigabytes of data that pass through tap points during any given execution, it is vital that string search be efficient in both time and space.

To satisfy these constraints, we developed `stringsearch`, a plugin which requires only one byte of memory per search string and per tap point. This one-byte counter tracks, for a given tap point, how many bytes of the search string have been matched by the data seen at the tap point so far. Whenever a byte is read from or written to memory, we can check what the next byte in the search string is using this position, and compare it to the byte passing through the tap point. If it matches, the counter is incremented; if it does not match, the counter is reset to zero. When the counter equals the length of the search string, we know that the search string has passed through the tap point, and we report a match. Note that because the counter is only one byte, our matcher only supports strings up to 256 bytes long; this cap could be easily raised to 65,536 bytes by using a two-byte counter, at the cost of doubling the memory requirements. Thus far, 256-byte strings have been more than sufficient.

This effectively implements a very simple deterministic finite automaton (DFA) matcher. Indeed, we believe that it should be possible to efficiently implement a streaming basic regular expression matcher that requires only an amount of memory logarithmic in the number of states needed to represent the expression. We leave this generalization to future

work, however.

4.5.4 Statistical Search and Clustering

Collecting bigram statistics on data that passes through each tap point is an efficient way to enable "fuzzy" search based on some training examples, as well as enabling clustering. To implement this we collect bigram statistics for all tap points seen in execution, as well as for the exemplar; the data seen at each tap point is thus represented as a sparse vector with 65,536 elements (one for each possible pair of bytes).

To search, we can then sort the tap points seen by taking the distance (according to some metric) from the exemplar. For our metric, we have chosen to use Jensen-Shannon divergence [74], which is a smoothed and symmetrized version of the classic Kullback-Leibler divergence [67] (also known as information gain). We also examined the Euclidean and cosine distance metrics, but found their performance to be consistently worse. Jensen-Shannon divergence between two probability distributions P and Q is defined as:

$$JSD(P,Q) = H\left(\frac{P+Q}{2}\right) - \frac{H(P)+H(Q)}{2}$$

where H is Shannon entropy.

Bigram collection is done by maintaining, for each tap point, two pieces of information: (1) the last byte that passed through the tap point, so that we can see bigrams that span a single memory access; (2) a histogram of all byte pairs seen at the tap point. The latter of these must be maintained sparsely: because our bigrams are based on bytes, a dense histogram would require 65,536 integers' worth of storage per tap point. Given that most of the executions examined in this chapter contain upwards of 500,000 tap points, this would require more than 120GB of memory, which is clearly infeasible (and wasteful, since most of those entries would be zero).

Instead, we store the histogram sparsely, using a C++ Standard Template Library `std::map<uint16_t,int>`. This keeps memory usage down without sacrificing any accuracy, but it does introduce some extra complexity when processing the resulting histograms, as our search software must support sparse vectors rather than simple arrays. Because of

this additional complexity, we opted to implement the search and clustering algorithms ourselves, after some initial prototyping using SciPy's `sklearn` toolkit.

Our clustering is based on the venerable k-means algorithm [117], but using the Jensen-Shannon divergence described in the previous section. As in the statistical search case, we use bigram statistics for our feature vectors. Initialization uses the KMeans++ algorithm [4], which helps guarantee that the initial cluster centers are widely separated. We evaluate the performance of this clustering compared to an expert labeling in Section 4.6.3.

Our statistical search tool is implemented in 246 lines of C++, and computes the Jensen-Shannon divergence between a training histogram (dense) and a set of sparse histograms. Our K-Means clustering tool is 481 lines of C++ code, and outputs a clustering of the sparse histograms using Jensen-Shannon divergence as a distance metric.[4] Both tools are multithreaded, which greatly speeds up the computation.

4.5.5 Finding SSL/TLS Keys

We have also implemented a PANDA plugin called `keyfind`, which locates tap points that write SSL/TLS master secrets. The SSL/TLS master secret is a 48-byte string from which an SSL/TLS-encrypted session's keys are derived; thus, if a tap point that writes the master key can be found, encrypted network traffic can be decrypted and analyzed.

The plugin operates on a recording in which a program initiates an encrypted connection to some server and an encrypted packet sent by the client (captured using, e.g., `tcpdump`). The `keyfind` uses each 48 bytes accessed at each tap point as a trial decryption key for a sample packet sent by the client. If the decrypted packet's Message Authentication Code (MAC) verifies that the packet was decrypted correctly then we can conclude that the tap point can be used to decrypt SSL/TLS connections made by the program under inspection. In Section 4.6.2.1 we show how this technique can be used to spy on connections made by the Sykipot malware, without performing a (potentially detectable) man in the middle attack.

[4]The use of this distance metric is justified theoretically because Jensen-Shannon distance is a Bregman divergence [8] and empirically because our clustering typically converges after around 30 iterations.

Table 5: Tap points found that write the URL typed into the browser by the user.

Browser	Caller	PC
Deb Epiphany (arm)	`WebCore::KURL::KURL+0x30`	`WebCore::KURL::init+0x70`
Deb Epiphany (amd64)	`webkit_frame_load_uri+0xc3`	`WebCore::KURL::init+0x368`
Win7 IE8 (x86)	`ieframe!CAddressEditBox::_Execute+0xaa`	`ieframe!StringCchCopyW+0x50`
Win7 Firefox (x86)	`xul!nsAutoString::nsAutoString+0x1a`	`xul!nsAString_internal::Assign+0x1d`
Win7 Chrome (x86)	`msftedit!CTxtEdit::OnTxChar+0x105`	`msftedit!CTxtSelection::PutChar+0xb8`
Win7 Opera (x86)	`Opera.dll+0x2cf6c6`	`Opera.dll+0x142783`
Haiku WebPositive (x86)	`BWebPage::LoadURL+0x3a`	`BMessage::AddString+0x26`

4.6 Evaluation

In this section, we evaluate the efficacy of our various tap point search strategies, described in Section 4.4, at finding tap points useful for introspection. Our experiments are motivated by real-world introspection applications, and so for each experiment we describe a typical application for the tap points found. Each experiment was also generally performed on a variety of different operating sytems, applications, and architectures in order to evaluate TZB's ability to handle a diverse range of introspection targets.

For the sake of readability, we have attempted to use symbolic names for addresses wherever possible in the following results. It is hoped that these will be more meaningful to the reader than the raw addresses, but we emphasize that debug information is in no way required for TZB to work.

4.6.1 Known Knowns

4.6.1.1 URL Access

Monitoring visited URLs is likely to be useful for host-based intrusion detection and prevention systems. For example, an IDS may wish to verify that outgoing requests were initiated by a human rather than malware on the users's machine, or match URLs visited against a blacklist of malicious sites. This poses a challenge for existing introspection solutions, as URL load notification is not generally exposed by a public API, and the data resides in a user application (the browser).

To find URL tap points, we created training executions by visiting a set of three URLs (Google, Facebook, and Bing) in the following operating systems and browsers: Epiphany on Debian squeeze (armel and amd64); Firefox 16.0.2, Opera 12.10, and Internet Explorer 8.0.7601.17514 on Windows 7 SP1 (x86); and WebPositive r580 on Haiku (x86). We used the `stringsearch` plugin to search for the ASCII and UTF-16 representations of the three

Table 6: Tap points found that write the SSL/TLS master secret for each SSL/TLS connection.

Client	Caller	PC	Process
Deb OpenSSL (arm)	`tls1_generate_master_secret+0x9c`	`tls1_PRF+0x90`	openssl
Deb OpenSSL (amd64)	`ssl3_send_client_key_exchange+0x437`	`tls1_generate_master_secret+0x108`	openssl
Deb Epiphany (arm)	`md_write+0x74`	`md5_write+0x68`	epiphany
Deb Epiphany (amd64)	`md_write+0x60`	`md5_write+0x49`	epiphany
Haiku WebPositive (x86)	`tls1_generate_master_secret+0x65`	`tls1_PRF+0x14b`	WebPositive
Win7 Chrome (x86)	`chrome!NSC_DeriveKey+0x1241`	`chrome!TLS_PRF+0xa0`	chrome.exe
Win7 IE8 (x86)	`ncrypt!Tls1ComputeMasterKey@32+0x57`	`ncrypt!PRF@40`	lsass.exe
Win7 Firefox (x86)	`softokn3!NSC_DeriveKey+0xe85`	`freebl3!TLS_PRF+0xbb`	firefox.exe
Win7 Opera (x86)	`Opera.dll+0x2eb06e`	`Opera.dll+0x50251`	opera.exe

URLs, and then validated each tap point found to ensure that it wrote only the desired data. The results can be seen in Table 5.

4.6.1.2 TLS/SSL Master Secrets

Monitoring SSL/TLS-encrypted traffic is a classic problem for intrusion detection systems. Currently, hypervisor- or network- based IDSes that wish to analyze encrypted traffic must perform a man-in-the-middle attack on the connection, presenting a false server certificate to the client. Not only does this require the client to cooperate by trusting certificates signed by the intrusion detection system, it also takes control of the certificate verification process out of the hands of the client—a dangerous step, given that many existing SSL/TLS interception proxies have a history of certificate trust vulnerabilities [51].

Instead of a man-in-the-middle attack, we can instead use TZB to find a tap point that reads or writes the SSL/TLS master secret for each encrypted connection, giving us a "man-on-the-inside". Because this secret must be generated for each SSL/TLS connection, if we can find such a tap point, it can then be provided to the IDS to decrypt and, if necessary, modify the content of the SSL stream.

To find the location of these tap points, we ran a modified copy of OpenSSL's `s_server` utility that prints out the SSL/TLS master key any time a connection is made. We then recorded executions in which we visited the server with each of our tested SSL clients, and noted the SSL/TLS master secret. Finally, we used `stringsearch` to search for a tap point that wrote the master key, and verified that the tap wrote exactly one master key per connection. For this test, we used: OpenSSL s_client 0.9.8 on Debian squeeze (armel), OpenSSL s_client 0.9.8 and Epiphany 2.30.6 on Debian squeeze (amd64), and Firefox 16.0.2, Google Chrome 23.0.1271.64, Opera 12.10, and Internet Explorer 8.0.7601 on Windows 7

Table 7: Tap points found for file access on different operating systems.

Target	Caller	PC
Debian (amd64)	`getname+0x13e`	`strncpy_from_user+0x52`
Debian (arm)	`getname+0x88`	`__strncpy_from_user+0x10`
Haiku (x86)	`EntryCache::Lookup+0x27`	`hash_hash_string+0x1b`
FreeBSD (x86)	`namei+0xd1`	`copyinstr+0x38`
Windows 7 (x86)	`ObpCaptureObjectName+0xcb`	`memcpy+0x33`

SP1 (x86). The results are shown in Table 6.

There is one particular point of interest to observe in these results. In the case of Epiphany on Debian, we found that one level of callstack information was *not* sufficient—with only the immediate caller, the tap point contains more data than just the SSL/TLS master secret. This is because the version of Epiphany uses SSLv3 to make connections, and the pseudo-random function (PRF) used in SSLv3 has the form

$$MD5(SHA1(\ldots))$$

The other implementations instead use TLSv1.0, where the PRF has the form

$$MD5(\ldots) \oplus SHA1(\ldots)$$

This final XOR operation is done from a unique program point, so the tap point that results from it contains only TLS master keys. This points to a potential complication of using tap points for introspection: it is not always clear in advance how many levels of call stack information will be required.

We were successful in locating tap points for all SSL/TLS clients tested. We note that uncovering similar information using traditional techniques would have required significant expertise and reverse engineering of both open source and proprietary software.

4.6.1.3 File Access

Monitoring file accesses is a requirement for many host-based security applications, including on-access anti-virus scanners. Thus, locating a tap point at which system-wide file accesses can be observed is of considerable importance. However, because previous approaches to the introspection problem [32, 40] passively retrieve information from the guest and are not event-driven, they cannot be used in this scenario.

To find such a tap point, we created recordings in which we opened files in various operating systems. Specifically, in each OS we created 100 files, each named after ten successive digits of π. The operating systems chosen for this test were: Debian squeeze (amd64), Debian squeeze (armel), Windows 7 SP1 32-bit, FreeBSD 9.0, and Haiku R1 Alpha 3 (all on x86). We then searched for tap points that wrote strings matching the ASCII and UTF-16 encodings of the filenames using the `stringsearch` analysis plugin. The UTF-16 encodings were included because it was known that Windows 7 uses UTF-16 for strings pervasively, allowing us to surmise that on Windows URLs would likely be UTF-16 encoded. Finally, we looked at the tap points found by `stringsearch`, and validated them by hand.

The results are shown in Table 7. For most of the operating systems we had no difficulty finding a tap point that contained the name of each file as it was accessed. The one exception was Windows 7, where the most promising tap point not only wrote file results, but also a number of unrelated objects such as registry key names. As in the SSL case, the root cause of this was insufficient calling context: in Windows several different things fall under the umbrella of a "named object", and these were all being captured at this tap point. We found that four levels of calling context were sufficient to restrict the tap point to just file accesses; the "deepest" caller was `IopCreateFile` (which, despite its name, is used for both opening existing files and creating new ones).

4.6.2 Known Unknowns

4.6.2.1 SSL Malware

The need to snoop on SSL-encrypted connections arises in malware analysis as well. Two features distinguish this case from that of intercepting the traffic of benign SSL clients presented in the previous section. First, the ability to decrypt the traffic without a man in the middle is even more important: in contrast to benign clients, we cannot assume that malware will accept certificates signed by our certificate authority. Second, we cannot rely on having access to the server's master secret, as the server is under the attacker's control. This means that our previous strategy of using a simple string search for the master secret

Table 8: Tap points that write the system log (`dmesg`) on several UNIX-like operating systems. All tap points were located in the kernel, except for Minix, which is a microkernel. We were unable to find a tap point analogous to `dmesg` in Windows.

OS	Caller	PC	Kernel?	Rank
FreeBSD (x86)	`msglogstr+0x28`	`msgbuf_addstr+0x19a`	Yes	1
Haiku (x86)	`ring_buffer_peek+0x59`	`memcpy_generic+0x14`	Yes	1
Debian (arm)	N/A	`do_syslog+0x18c`	Yes	4
Debian (amd64)	N/A	`do_syslog+0x163`	Yes	4
Minix (x86)	`0x190005ee`	`0x190009d4`	No	8
Windows 7 (x86)	Not Found	Not Found	?	?

will not work here.

Instead, we located the tap point in the SSL-enabled malware using our `keyfind` plugin, which performs trial decryption on a packet sent by the malware using each possible 48-byte sequence written to memory as a key and verifies whether the Message Authentication Code is valid. Although this is much slower than a string match, it is the only available option, since the key is not known in advance.

To test the plugin, we obtained a copy of a version of the Sykipot trojan released around October 31st, 2012 [44] (MD5: `34a1010846c0502f490f17b66fb05a12`). We then created a recording in which we executed the malware; simultaneously, we captured network traffic using `tcpdump`. We noted that the malware made several encrypted connections to `https://www.hi-techsolutions.org/`, and provided one of the encrypted packets from these connections as input to the `keyfind` plugin. The plugin found the same tap point as the Windows 7 IE8 experiment described earlier, indicating that both the malware and IE8 likely use the same underlying system mechanism to make SSL connections. The key found was able to decrypt the connections contained in the packet dump.[5]

4.6.2.2 Finding `dmesg`

System logs are an invaluable resource, both for security and system administration. In an introspection-based security system, for example, one might want to find a tap point that contains the system's logs so that they can be stored securely outside the guest virtual machine. However, because the format of system logs is particular to each OS, we need

[5]The malware also has a second layer of encryption, which is custom and not based on SSL; we did not attempt to decrypt this second layer.

some mechanism that can find tap points that write data that "looks like" a log based on an exemplar. The statistical search described in Section 4.4.2 is a good fit for this task: by training on the output of `dmesg` on one OS, we can find `dmesg`-like tap points on other systems.

To locate these system log tap points, we first created a training exemplar by running the `dmesg` command on a Debian sid (amd64) host and computing the bigram probabilities for the output. We then created recordings in which we booted five operating systems (Debian squeeze (armel), Debian squeeze (amd64), Minix R3-2.0 (i386), FreeBSD 9.0-RELEASE (i386), and Haiku R1 Alpha 3 (i386)), and computed the same bigram statistics. We then sorted the tap points seen in each operating system boot according to their Jensen-Shannon distance from the training distribution, and manually examined data written by the tap point for each of the top 30 results in each operating system. Table 8 shows, for each operating system tested, the tap point that we determined to be the system log, and its rank in the search results.

We can see that in all cases the correct result is in the top 10. There are two additional features of Table 8 that bear mentioning. First, the reader will note that the two Debian systems have a caller of "N/A". This is because the memory writes that make up `dmesg` are done in `do_syslog`, which is called from multiple functions. In these cases, including the caller splits up information that is semantically the same. We detected this case by noticing that several of the top-ranked results in the Linux experiments had the same program counter, and that they appeared to contain different sections of the same log. Second, the tap point found for Haiku was also incomplete—some lines were truncated. By using our tap point correlation plugin, we determined that we were missing a second tap point that was correlated with the main one; the two together formed the write portions of a `memcpy` of the log messages. Once this second tap point was included, we could see all the log messages produced by Haiku.

We also attempted to find an analogous log message tap point on Windows 7, but were not successful. This is a result of the way Windows logging works: rather than logging string-based messages, applications and system services create a manifest declaring possible

log events, and then refer to them by a generated numeric code. Human-readable messages are not stored, and instead are generated when the user views the log. This means that there is no tap point that will contain log messages of the type used in our `dmesg` training, and the methods described in this chapter are largely inapplicable unless a training example for the binary format can be found. However, because the event log query API is public [82], existing tools such as Virtuoso [32] might be a better fit for this use case.

Anecdotally, the ability to uncover a tap point that writes the kernel logs has also been useful for diagnosing problems when adding support for new platforms to QEMU. For an unrelated research task, we attempted to boot the Raspberry Pi [1] kernel inside QEMU, but found that it hung without displaying any output early on in the boot process. By locating the `dmesg` tap point, we discovered that the last log message printed was "Calibrating delay loop..."; based on this we determined that the guest was hung waiting for a timer interrupt that was not yet implemented in QEMU.

4.6.3 Unknown Unknowns: Clustering

To test the effectiveness of clustering tap points based on bigram statistics and Jensen-Shannon distance, we carried out an experiment that compared the clusters generated algorithmically to a set of labels generated by two of the co-authors manually examining the data. We created six recordings representing different workloads on two operating systems (Windows 7 and FreeBSD 9.0). From FreeBSD, we took recordings of boot, shutdown, running applications (`ps`, `cat`, `ls`, `top`, and `vi`), and a one-minute recording of the system sitting idle, for a total of four recordings. On Windows we created two recordings: running applications (`cmd.exe`, `dir`, the Task Manager, Notepad), and one minute of the system sitting idle.[6]

Next, we sampled a subset of the tap points found in each recording. Given that the vast majority of tap points do not write interesting information, we opted not to sample uniformly from the all tap points found. Instead, we performed an initial k-means clustering with $k = 100$, and then picked out tap points at various distances from each

[6] Although we would have preferred to include Windows boot and shutdown recordings, at the time our replay system had a bug (now fixed) that prevented these recordings from being replayed.

cluster center. We chose the tap point at σ standard deviations from the center, for $\sigma \in \{0, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0\}$ for a total of 2,926 samples.[7] Finally, we dumped the data from each of the sampled tap points, blinded them by assigning each a unique id, and then provided the data files to our two labelers. Each labeler independently assigned labels to each of the samples using the labels described in Table 9, and the two labelers then worked together to reconcile their labels.

Finally, we ran a k-means clustering with $k = 10$; 10 was chosen because it was a round number reasonably close to the number of labels our human evaluators gave to the data. We then used the Adjusted Rand Index [47] to score the quality of our clustering relative to our hand-labeled examples. The Adjusted Rand Index for a clustering ranges from -1 to 1; clusterings which are independent of the hand labeling will receive a score that is negative or close to zero. As can be seen in Table 10, our clustering did not match up very well on the hand-labeled samples. Note, however, that labeling criteria were selected without knowledge of the sizes of categories or whether or not the distance metric would effectively discriminate, so it is perhaps unsurprising that the correspondence is poor.

There is some hope, however. Regardless of the apparently poor clustering performance with respect to hand-labeling, we decided to determine if the clusters from our 100-mean clustering of FreeBSD's boot process contained new and interesting data and if finding that data would be facilitated by them. First, we determined to which of the clusters data from the FreeBSD's `dmesg` and filename tap points (found in Sections 4.6.2.2 and 4.6.1.3) was assigned. We were heartened to learn that these two text-like tap points had been sent to the same cluster. We proceeded to explore this cluster of approximately 5000 tap points, and found that, indeed, the vast majority of the tap points contained readable text of some sort. Further, in the course of about thirty minutes of spelunking around this cluster, we found not only kernel messages and filenames but a stone soup of shell scripts, process listings kernel configuration, GraphViz data, and so on. A selection of these tap point contents is provided in Appendix A. We did not exhaustively examine this cluster, but plan to do

[7] The alert reader will note that this is smaller than the 4,800 samples one would expect from taking 8 samples from 100 clusters in each of 6 recordings. This is because some clusters did not have very high variance, and so in many cases there were fewer than 8 samples at the required distance from the center.

Table 9: Labels given to the sampled tap points by human evaluators, along with the number of times each occurred.

Abbrv.	Description	Count
bp	binary pattern	2318
rd	repeated dword	400
mz	mostly zero	141
rq	repeated quadword	19
fnu	filenames unicode	8
woa	words ascii	8
wou	words unicode	7
inu	integers unicode	6
bu	binary uniform	5
ura	URLs ascii	5
rs	repeated short	4
fna	filenames ascii	2
rb	repeated byte	2
vr	very redundant	1

Table 10: Quality of clustering as measured by the Adjusted Rand Index, which ranges from -1 to 1, with 1 being a clustering that perfectly matches the hand-labeled examples.

Recording	ARI
FreeBSD Apps	0.018
FreeBSD Boot	0.048
FreeBSD Idle	0.021
FreeBSD Shutdown	0.074
Win7 Apps	0.029
Win7 Idle	-0.003

so soon, as it appears to contain much of interest for active monitoring. If clustering has focused us on one out of 100 clusters, this is potentially a big savings.

4.6.4 Accuracy

Leaving aside the clustering results for the moment, the analyses implemented in TZB are extremely effective at helping to identify interposition points for active monitoring. In the evaluations based on string searching, we found that the number of tap points we had to look at manually was at most 262 (URLs under IE8) and in the best case we only had to examine two tap points (for SSL keys under Firefox, Opera, Haiku, and OpenSSL on ARM). The number of tap points that need to be examined is related to how widely the data is propagated in the system and how common the string being searched for is; thus, it is natural that URLs visited in the browser would appear in many tap points, whereas the SSL/TLS master key would not. Qualitatively speaking, we found that once the candidate

tap points had been selected by `stringsearch` for a given execution, it took at most an hour to find one that sufficed for the task at hand.

For the `dmesg` evaluation, we also examined the quality of the results found for each operating system using the standard "Precision at 10" metric, which is just the number of results found in the top 10 that were actually relevant to the query. In this case, this is simply the fraction of results in the top 10 that appeared to contain the system log (even if it was incomplete). Based on this metric, the precision of our retrieval was between 20% (on Minix) and 100% (on Haiku). This means that if one looked at all of the top 10 entries, it is guaranteed that one would find the correct tap point.

4.7 Limitations and Future Work

Although TZB is currently very useful for finding interception points for *active monitoring*, it is not currently usable in every scenario where introspection is needed. Because the interception points are triggered by executing code, they are only usable in *online* analysis. However, the need for introspection also arises in *post-mortem* analysis, specifically in forensic memory analysis. Whereas previous solutions such as Virtuoso [32] were able to operate equally well on memory images or live virtual machines, TZB is only applicable to the live case. In future work, we hope to combine Virtuoso-like techniques with TZB to produce offline programs that can locate in memory the buffers on which TZB's tap points operate.

Another limitation of TZB is its reliance on callstack information to locate interposition points. In current systems, keeping track of an arbitrary number of callers for each process is prohibitively expensive; although stack walking is faster (since it only needs to be invoked when the monitored code is executed), it is insecure, unreliable, and not available on every architecture. We hope to examine how existing solutions such as compiler modifications [23, 122] or dynamic binary translation [108] can be used to efficiently maintain the shadow stack information needed for TZB. We also note that Intel's Haswell architecture, which as of this writing has just been released, has hardware support for keeping track of calls and returns [129]; this would provide an excellent base on which to build a low-overhead

security system based on TZB tap points.

Code generated at runtime (i.e., JIT or injected code) may make re-identification of a discovered tap point difficult or even impossible. Given the rise of languages that depend on JIT runtimes, better solutions are needed for this scenario, and the problem of how to make use of tap points in JIT code should be explored further.

Finally, as seen in Section 4.6.3, the clustering results are promising, but not yet fully developed. We hope to gain a better understanding of the data found in tap points and seek out better features and models for clustering in future work.

4.8 Related Work

Although, to our knowledge, there is no existing work on mining the contents of memory accesses for introspection, we drew inspiration from a variety of sources. These can be roughly grouped into three categories: work on automating virtual machine introspection, research on automated reverse engineering, and efforts that examine memory access patterns, typically through visualization. In this section, we describe in more detail previous work in these three areas.

Virtual machine introspection has been targeted for automation by several recent research efforts because of the *semantic gap problem*: security applications running outside the guest virtual machine need to reconstruct high-level information from low-level data sources, but doing so requires knowledge of internal data structures and algorithms that is costly to acquire and maintain. To address this problem, researchers have sought ways of bridging this gap automatically. Virtuoso [32] uses dynamic traces of in-guest programs to extract out-of-guest tools that compute the same information. However, because it is based on dynamic analysis, incomplete training may cause the generated programs to malfunction. Two related approaches attempt to address this limitation: *process out-grafting* [115] moves monitored processes to the security VM while redirecting their system calls to the guest VM, allowing tools in the security VM to directly examine the process, while VMST [40] selectively redirects the memory accesses of tools like `ps` and `netstat` from the security VM so that their results are obtained from the guest VM. TZB extends these approaches

by finding points in applications and the OS at which to perform active monitoring.

Based on the observation that memory accesses in dynamic execution can reveal the structure of data in memory, several papers have proposed methods for automatically deducing the structure of protocols [19, 28, 75], file formats [29, 76], and in-memory data structures [70, 77, 110]. One particular insight we have drawn from this body of work is the idea that the point in a program at which a piece of data is accessed, along with its calling context, can be used as a proxy for determining the type of the data. TZB leverages this insight to separate out memory accesses into streams of related data.

Finally, there has been some research on examining memory accesses made by a single program or a whole system, typically using visualization. Burzstein et al. [16] found that by visualizing the memory of online strategy games, they could identify the region of memory used to decide how much of the in-game map was visible to the player, which greatly reduces the work required to create a "map hack" and allow the player to see the entire map at once. Outside the academic world, the ICU64 visual debugger [80] allows users to visualize and modify the entire memory of a Commodore 64 system, enabling a variety of cheats and enhancements to C64 games. Although TZB does not use visualization, it shares with this previous work the understanding that memory accesses can be a rich source of information about a running program.

4.9 Conclusion

In this chapter we have presented TZB, a system that automatically locates candidate memory accesses for active monitoring of applications or operating systems. This is a task that previously required extensive reverse engineering by domain experts. We have successfully used TZB to identify a broad range of tap points, including ones to dynamically extract SSL keys, URLs typed into browsers, and the names of files being opened. TZB is built atop the QEMU-based PANDA platform as a set of plug-ins and its operation is operating system and architecture agnostic, affording it impressive scope for application. This is a powerful technique that has already transformed how the authors perform RE tasks. By reframing a difficult RE task as a principled search through streaming data

provided by dynamic analysis, TZB allows manual effort to be refocused on more critical and less automatable tasks like validation.

CHAPTER V

IDENTIFYING FUNCTIONALITY IN PREVIOUSLY-UNSEEN MALWARE

5.1 Motivation

When faced with a new malware sample, an analyst's goal is typically to quickly develop a high-level understanding of its functionality and capabilities. Typically, this is done through a combination of manual static analysis (i.e., analyzing a disassembly of the malicious binary by hand) or by some kind of behavioral analysis (e.g., running the malware in a sandbox and examining its interactions with the outside world—files created, network connections made, etc). The former is extraordinarily time consuming and requires significant expertise but can produce a detailed and complete understanding of the malicious code; on the other hand, a behavioral analysis is trivial to carry out and requires very little time but is unlikely to give a full picture of the malicious behavior.

In this chapter we propose a middle ground that attempts to provide a high-level picture of an unknown binary's functionality based on dynamic features of its execution. Specifically, for each function in an unknown binary, we will attempt to find the closest-matching function in our library of previously-seen functionality. An important distinction here is that we would like our algorithm to perform well even if no exact match can be located; that is, we hope to *match similar functionality across different binaries.* In the general case, of course, performing this task is impossible—deciding the equivalence of two functions is equivalent to the halting problem. However, our work follows in the grand tradition of tackling undecidable problems by sacrificing certainty and instead using heuristics and looking for approximate matches.

Our overall approach is depicted in Figure 22. First, a collection of labeled training examples is gathered; these training examples are programs whose functionality lies in the same domain as the samples we will wish to classify later (for example, when looking at

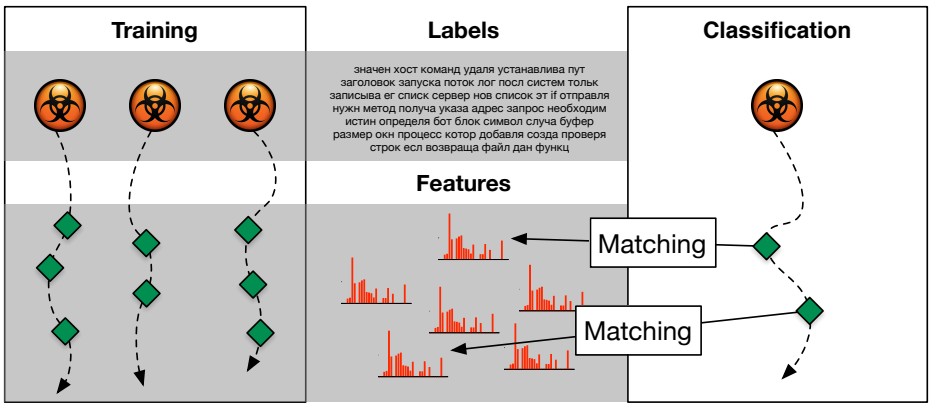


Figure 22: The design of our function matching system. Training samples are executed, producing dynamic features; at the same time, labels are statically extracted and associated with the dynamic features. Finally, to label a new sample, the sample is executed and the same features are collected. The new sample can then be labelled by looking at the labels given to the closest-matching features in the training set.

functionality in malware samples one would want a labeled corpus of malware to train on). Next, the training programs are run, and dynamic features of their execution are gathered; this associates each labeled function with a set of dynamic features. Finally, to identify functionality in an unknown program, we run it and collect the same dynamic features as in the training set; these features are then compared with those from the functions in the training set and the closest matches are reported.

Aside from malware analysis, we believe the techniques described in this chapter may also be useful in a number of other areas. When auditing closed-source software for vulnerabilities, for example, researchers often must locate code that is more likely to contain bugs, such as parsing or authentication code; our techniques could help quickly narrow down which functions deserve manual analysis.

5.2 Related Work

Our work seeks to recover high-level information about binary executions by comparing unknown functions' execution with a library of known and labeled executions in order to automatically label behaviors in unknown malware. Although, to the best of our knowledge, our technique and application is novel, there is a great deal of work related to each component.

An enormous body of work exists on analyzing and detecting malware. Static approaches such as semantics-aware malware detection [24] attempt to demonstrate the equivalence of two functions using techniques from compiler theory and thereby establish that a function is derived from a previously seen malicious function. BitShred [49] and iLine [50] also use static features to identify previously seen functionality; in the former case a Bloom filter of byte sequences is used, whereas the latter looks at a variety of static features such as code section size, file size, and cyclomatic complexity. While the use of such static features can allow analysis of large volumes of software (since it is not necessary to execute the software and collect dynamic features), static techniques are typically not robust to transformations that preserve functionality while radically changing the structure of the code.

Other approaches have sought to use high-level dynamic behaviors for malware detection, either by using system calls as a proxy for behavior [71] or looking at whole-system changes in state [6]. Such work has typically treated the problem as one of machine learning classification, where an unknown program as a whole is to be classified as malicious or non-malicious; for example, Rieck et al. combines clustering and behavioral classification based on a vector representation of the actions taken by a malware sample. While we believe our techniques may serve as useful input to such a system, we are concerned with the more fundamental and fine-grained problem of identifying the behavior of individual functions in malicious and benign programs.

Closest to our own work appears to be the work of Egele et al. [34], which is due to appear at the 2014 USENIX Security Symposium but is, at the time of this writing, not available. Their technique, dubbed Blanket Execution (BLEX), forces each function to be evaluated in a controlled, randomized environment and logs its side effects. These side effects are then used to match unknown functions against previously seen ones. The authors report that the correct result for function matches appeared in the top ten results 77% of the time across all programs tested; although it is difficult to tell from just the abstract of the paper, it appears that our techniques significantly outperform BLEX.

5.3 Design and Implementation

In this section we describe the design and implementation of our system for performing functionality matching using dynamic features.

5.3.1 Label Extractor

Our current system assumes that source code is available for the programs in the training set. Given that, we wish to automatically extract labels that describe each function in each binary in our training set. If source code is not available, it would also be possible in principle to have a human analyst label the functions in the training set; while this would be extremely time consuming, it might actually produce a more accurate labeling.

Since our focus is on malware, which is seen in greatest abundance on the Windows platform, we developed a label extractor based on debugging information produced by Visual Studio, i.e. PDB files. For each binary, we use the Debug Interface Access SDK [81] to extract the local variables associated with each function, and to create a mapping between each function and the source lines it was compiled from. We then use a simple Python script to examine the source of each function and extract any nearby comments and associate them with the function; for simplicity, we treat the comments a simple set of words.

Because we expect comments to be written in natural language, we additionally normalize comments by performing *stemming* (normalizing each word so that inflected and derived forms are reduced to a common base) and *stopword removal* (removing common words that provide no descriptive value; in English examples are "the", "and", "of", etc.). Stemming and stopword removal are implemented using the Python NLTK package [14]. Because it is common, particularly in technical writing, to use a mix of the author's native language and English when writing comments, we attempt to automatically detect the language of each word before applying the appropriate stemmer and stopword list. For language detection we use the `langid.py` package [79]; however, this process is still highly inaccurate given the difficulty of identifying a language from just one word. We hope to improve on this in future work.

The local variable and source line extractor consists of 60 lines of C# code. The comment

extractor consists of 133 lines of Python code, including natural language normalization. The final output is a serialized Python dictionary mapping function names to the words and variable names that describe the function.

5.3.2 Feature Extractors

Feature extraction is perhaps the most important component of the system, as it provides the basis for the matching. All features used are *dynamic*: they are obtained by actually running each program. The intent is that these features will be robust to simple structural changes to the code such as obfuscation, compiler optimizations, and so on. We also hope that it will be able to detect similarities in two different *implementations* of the same algorithm.

We currently extract two features: bigram frequencies for bytes read and written by each function, and the sequences of system calls made by each function and its callees. We discuss each in detail in this section. Feature extraction is performed using the PANDA dynamic analysis platform, described in Section 4.5.1. Both training programs and programs to be analyzed are run inside PANDA to generate a recording; feature extraction is then performed by replaying the executions with the feature extraction plugins described below enabled. As an optimization, feature extraction plugins only extract features from a pre-specified list of relevant processes.

5.3.2.1 Bigram Frequencies

As each function executes, it will read and write memory. As in TZB (see Chapter 4 for details), we can create a histogram of the content of memory reads and writes by instrumenting every read and write in the system. Unlike with TZB, we do not group each read and write by a tap point; instead, we group together the reads and writes made throughout the entire function. Thus for each function we will compute two histograms: one for reads and one for writes. Histograms are stored as sparse arrays in a binary log file for later analysis. The `bigrams` plugin is implemented in 207 lines of C++ code.

Because functions for which we observed very little data are unlikely to have a strong signature that can be used for a matching, we currently exclude any function whose reads

and writes were each less than 80 bytes. This parameter was determined more or less arbitrarily, and further experimentation is needed to determine the optimal threshold. In practice, we have had good results with the current value.

5.3.2.2 System Calls

Another important dynamic feature is the sequence of system calls each function executes. Because system calls are the only way a binary can effect persistent changes to the state of the system, virtually every program of interest will make system calls. These calls may therefore be useful in identifying similar functions across different programs.

For each function, we record not only the system calls made at top level, but also those made by its callees. For example, suppose that a function F_1 executes $\{sc_1, sc_2\}$ and then calls F_2, which executes $\{sc_3, sc_4\}$ and then returns to F_1, which finishes by executing $\{sc_5\}$. In this case, the calls recorded for F_1 would be $\{sc_1, sc_2, sc_3, sc_4, sc_5\}$, while the calls recorded for F_2 would be just $\{sc_3, sc_4\}$.

In PANDA, this is implemented using a combination of the `sc_chain`, `syscalls`, and `callstack` plugins. The main feature extraction is implemented in `sc_chain`, which registers a callback in `syscalls` to be notified every time a system call executes. When it does, the `callstack` plugin is called to get the current call stack, and the current system call number is logged for every function in the stack; the list of system calls made so far for each function is kept in a working set until the function returns. In addition to the function address, a counter is used to keep recursive calls to the same function separate. The `sc_chain` plugin also registers a callback in the `callstack` plugin in order to receive a notification whenever a function returns. When a function returns, `sc_chain` removes its entry from the working set and writes out the observed system call sequence to a log file. The log file produced by the plugin, then, will contain one entry for each time the function executed, and each entry will list all system calls seen during its execution.

All three plugins are written in C++. The `sc_chain` plugin is 184 lines of code, and the `syscalls` and `callstack` plugins (which are a part of PANDA and not developed as part of this current work) are 1006 lines and 325 lines of code, respectively.

5.3.3 Feature Matching

Feature matching is accomplished by simply taking the features from each function observed in the execution of the target program and computing a distance measure to each function in the training set. We can then look at the top N functions (usually 10) and investigate whether they are functionally similar to our target function. We can also then apply the labels given to the top matches from our training set to the unknown function, potentially giving a high-level description of its functionality.

The complexity of this pairwise distance computation is $O(kn)$, where k is the number of functions in the target program and n is the number of functions in the training set. Because this can be quite intensive, we hope to use techniques from information retrieval in future work to speed up the search process.

For the bigram histogram distance computation, we use the same Jensen-Shannon described in Section 4.5.4. To measure the distance between sequences of system calls, we currently use Levenshtein [72] distance.

5.4 Experiments

5.4.1 Carberp

To explore the effectiveness of our chosen features in identifying common functionality, we performed an experiment with real-world malware. Although this is in general a significant challenge, as most malware does not come pre-labeled, we were able to take advantage of the fact that in recent years the source code of some malware families has been leaked. Specifically, we chose to examine the Carberp malware family, which was leaked in June 2013 [64].

The Carberp source code leak consists of a large number of different modules, including the code to several other malware families. The main bot projects, which are contained in the `BJWJ` (for "Black Joe White Joe") are primarily banking trojans; however, they can be extended through plugin functionality to support VNC backdoors, Bitcoin mining, and distributed denial of service [56]. There are several flavors of the bot available in the main

project, which appear to share a common core but have slightly different features and functionality. For our experiments, we chose to train on the `RU_Az` project and test on the `Full` project. As an example of the differing behavior between bot variants, we observed that the `RU_AZ` sample hooked several system APIs (NtClose, NtDeviceIoControlFile, NtQueryDirectoryFile, and NtResumeThread), whereas `Full` did not hook any APIs but did inject itself into `explorer.exe`.

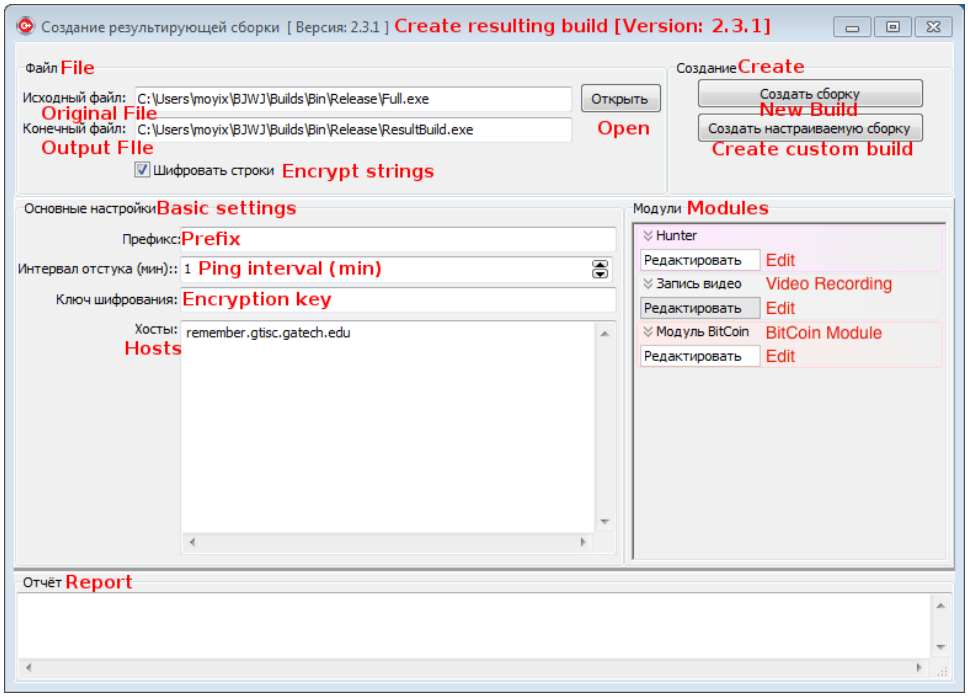


Figure 23: The Carberp builder UI, which injects an encrypted configuration section into a binary. Translations provided by the author with help from Sarah Montgomery and Google Translate.

Building Carberp from the leaked sources is not trivial; the Visual Studio project files provided are apparently incomplete and required a number of manual changes (primarily adding missing source files) before a successful build could be generated. Each project was built in its "Release" configuration, and we modified the build process so that debugging symbols would be generated in PDB format (as required by the label extractor described in Section 5.3.1). Once the initial binary is built, it must have a specific configuration added to it using a "builder" program (shown in Figure 23); this essentially injects an RC4-encrypted section into the executable that can then be decrypted and read when the bot starts up.

The configuration contains variables such as the host to contact for command and control (C&C), the interval to wait when contacting the C&C servers, the encryption key used for communication, and plugin-specific configuration options (for example, the host to submit Bitcoin mining results to).

We configured our Carberp binaries to connect to a host under our control, but did not set up any C&C infrastructure; as a result, it is possible that we did not observe the full range of behaviors present in the malware. However, the problem of obtaining good coverage of malware is a both well-studied [26,88] and orthogonal to the problem of function matching, so we will not explore it further in this work.

5.4.1.1 Feature Extraction and Labeling

We extracted labels for each sample as described in Section 5.3.2. Because Carberp's source code comments are in Russian, we used the Python NLTK package (specifically the Snowball stemmer) to perform word stemming and stopword removal; no modifications were needed to the variable names since the authors used English. The end result of this process was a mapping between a function name and a bag of words describing that function.

To obtain our features, we created a recording of each sample using PANDA. Each sample was executed, and we then waited ten minutes to allow time for the infection and initial communication to occur. We then replayed each with a custom process monitoring plugin that tracked the current process, as well as trapping calls to `NtCreateUserProcess` and `NtTerminateProcess`. This allowed us to determine which processes were involved in the execution of each sample, which we used to guide the actual feature extraction. For `RU_Az`, only the initial `RU_AzBuild.exe` process is used. In `Full`, an `explorer.exe` process is created, into which the bot binary is injected; execution then continues in the new process. We also observed both samples attempting to connect to a command and control server (our own) using the communication protocol documented by Gegeny [43].

We extracted the dynamic features described in Section 5.3.2 using the `bigrams` and `sc_chain` plugin; the former collects byte bigram statistics for memory read and written

by each basic block, while the latter logs the system calls executed by each function (including those executed in called subroutines). These plugins are described in more detail in Section 5.3.2.

5.4.1.2 Evaluation

Because are fortunate enough to have two Carberp builds that share a common core (see Figure 24, we get a certain measure of ground truth for free. Since a both binaries have a number of functions in common, we can evaluate whether our matching algorithm correctly determines that these functions are the same in two different executions.

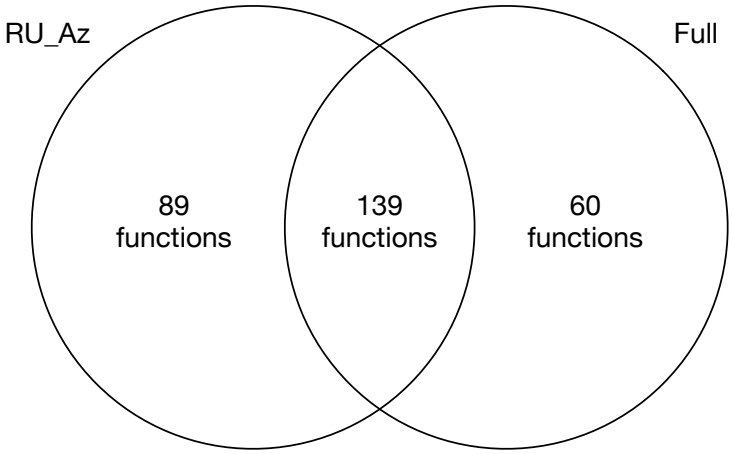
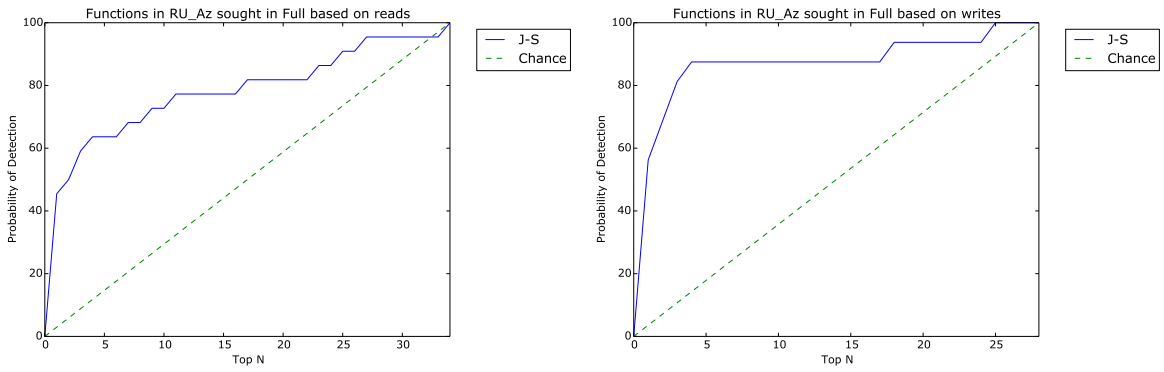


Figure 24: Code overlap in Carberp variants. The `RU_Az` and `Full` variants display different runtime behavior but share a common core of 139 functions.

Using the Jensen-Shannon metric described in Section 5.3.3, we scored each function in `RU_Az` against each function in `Full`; in other words, we used `Full` as our training set and evaluated our performance on `RU_Az`. For each function that was executed in both programs, we checked whether it appeared in the top N closest matches according to our metric. By varying N, we can then produce an ROC curve demonstrating our performance at various thresholds. Because writes and reads are tracked separately, we evaluated each. The results are shown in Figure 25.



(a) Carberp, read bigrams

(b) Carberp, write bigrams

Figure 25: Probability of detection for the top N ranked functions, based on the Jensen-Shannon distance between the bigram frequencies of data read and written by each function.

5.4.2 Zeus

Zeus [118] is another widely-distributed banking trojan. Like Carberp, its source has been leaked to the internet [66] and is available on GitHub [3], allowing us to compile the bot with debug information and run it. Unlike Carberp, compilation was relatively simple, although a few typos in the source code had to be corrected. Zeus also provides its builder UI and manual in English as well as Russian, obviating the need to translate. We modified its (PHP-based) build process to generate debugging symbols, and built three variants: `ZeusFull`, which was configured with all available features; `ZeusFullFast`, which was configured with the same features as `ZeusFull` but compiled with optimization for speed rather than size (`/O2` rather than `/O1` in Visual Studio), and `ZeusFullDebug`, which was built with Zeus's debug flag enabled, causing the bot to write log messages to a file.

We also installed the Zeus C&C server, which consists of a set of PHP scripts backed by a MySQL database. The bots were configured using the Zeus builder and set to contact our C&C server (depicted in Figure 26).

5.4.2.1 Feature Extraction and Labeling

Feature extraction and labeling is slightly more complicated for Zeus than Carberp because Zeus injects itself into every other process on the system upon execution. This allows it to implement a purely usermode "rootkit" by hooking the usermode calls to system

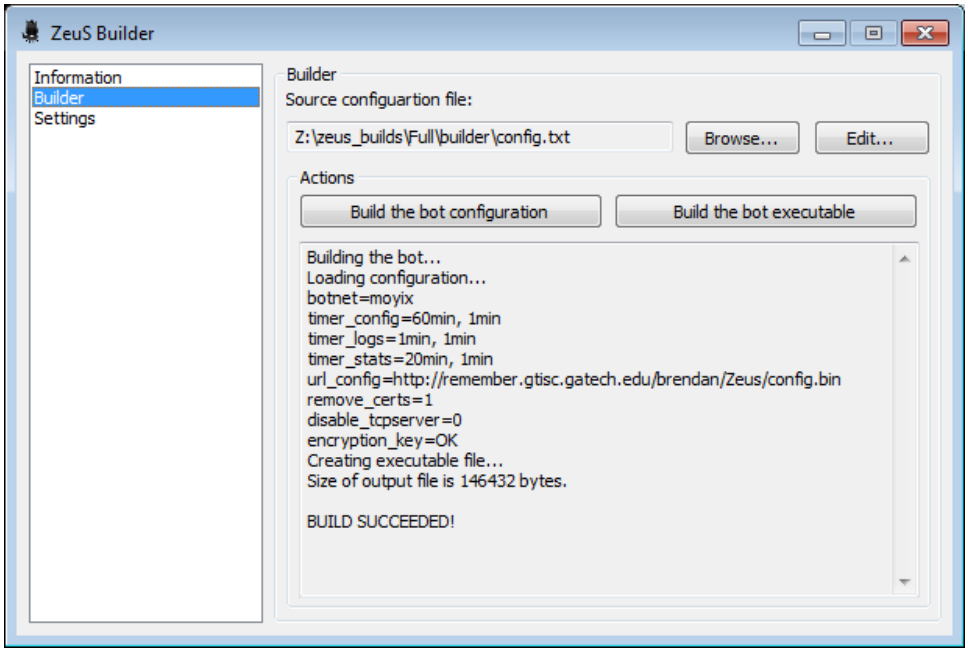


Figure 26: The Zeus builder UI, configured to inject our own C&C server into the bot binary.

functions rather than the kernel functions that implement those calls. However, it also means that there are a significantly larger number of processes to track when performing feature collection. To enable this, we developed a Volatility [124] plugin that searches for processes containing our Zeus binary, and modified our process monitoring plugin to create a memory dump just before any process exits. By doing so, we can compile a full list of processes relevant to the malware execution, as well as the ranges in memory where the code exists, even if it has been injected into another process.

Labeling is also complicated; rather than simply writing comments in either Russian or English, the authors of Zeus use a mix of the two, often mixing in Russian words into English sentences and vice versa.[1] Russian words are also present in both the standard CP1251 encoding and transliterated into the Roman alphabet. Stopword removal can be accomplished by just removing stopwords for both languages; however, stemming requires that we detect the correct language on a word-by-word basis and apply the appropriate stemmer.

[1] In linguistics, this is referred to as *code switching*.

Each replay was created by following the same procedure:

1. Download bot binary to the desktop inside PANDA virtual machine.
2. Begin recording.
3. Allow bot network connections through the firewall.
4. Wait for the bot to delete itself from the desktop.
5. Start Internet Explorer, and
   - Visit `http://google.com`.
   - Visit `http://vk.com`, a popular Russian social networking site.
6. Close Internet Explorer.
7. End the recording.

The web sites visited in Internet Explorer are an attempt to elicit more behaviors from the malware, as it contains code to perform man-in-the-browser attacks on popular web sites.

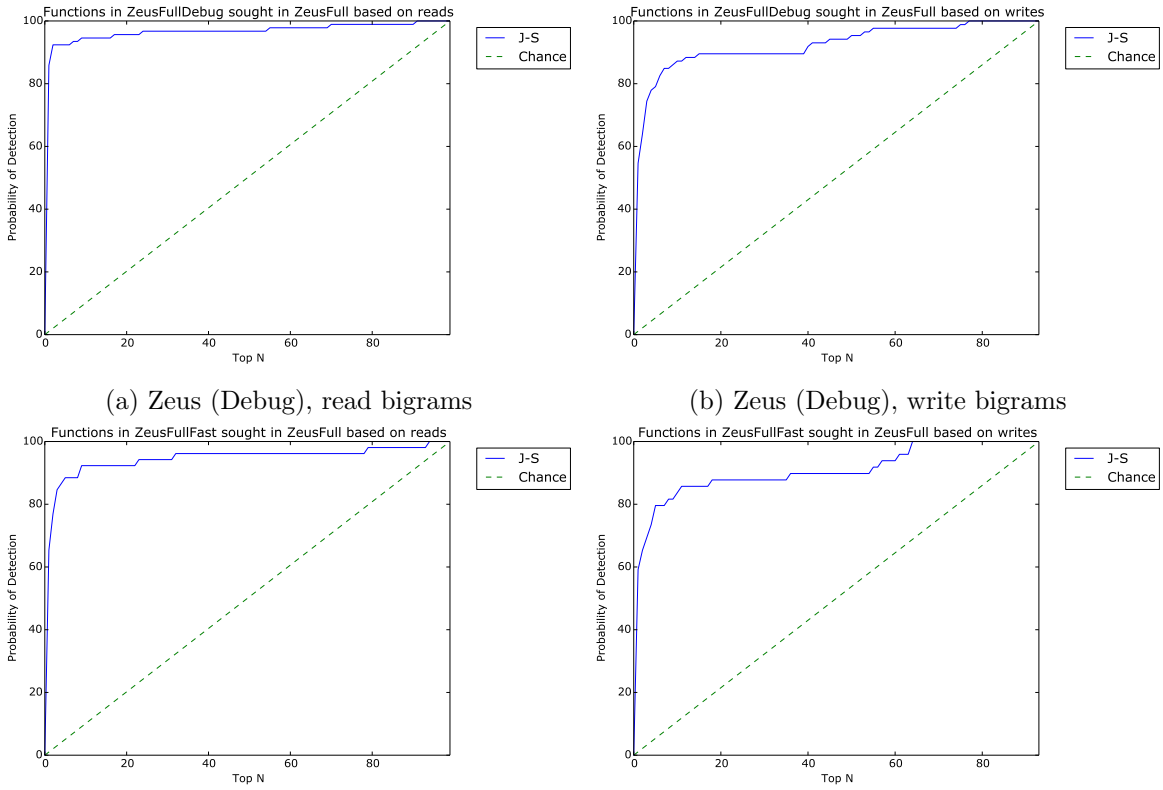
We then replayed each recording and collected the same features as for Carberp.

5.4.2.2 Evaluation

In the case of Zeus we know that all three execution traces are built from the same source code, but with different optimization and debugging options. They should therefore have approximately the same behavior making a close matching feasible. In our experiment, we attempted to identify functions in `ZeusFullFast` and `ZeusFullDebug` by matching them against functions from the basic `ZeusFull`. As before, ROC curves for both reads and writes are shown.

5.4.3 Discussion

To put these results in perspective, we can look at a related work by Egele et al. [34]; their system, BLEX (blanket execution), matches functions based on their dynamic side effects



(a) Zeus (Debug), read bigrams

(b) Zeus (Debug), write bigrams

(c) Zeus (Optimized), read bigrams

(d) Zeus (Optimized), write bigrams

Figure 27: Probability of detection for the top N ranked functions, based on the Jensen-Shannon distance between the bigram frequencies of data read by each function. The functions to be identified were drawn from the debug ((a) and (b)) and optimized ((c) and (d)) versions of Zeus and matched against the plain Zeus binary.

Table 11: The effect of requiring a minimum threshold of data before including a function's reads and writes. Figures given are the fraction of functions in the test binary that were correctly matched in the top 10 in the training set.

	Reads		Writes	
	No Threshold	Threshold	No Threshold	Threshold
`RU_Az` in `Full`	39%	72%	54%	88%
`ZeusFullDebug` in `ZeusFull`	90%	95%	65%	87%
`ZeusFullFast` in `ZeusFull`	83%	92%	67%	84%

and achieves high coverage through forced execution. Over all functions tested, their blanket execution system had an accuracy at $N = 10$ of 77%. Based on the abstract provided (the authors unfortuntaely declined to provide a preprint upon request), it appears that our Zeus results show significatnly better performance than BLEX at $N = 10$ at 92-95%.

Our choice to discard functions with too little data, described in Section 5.3.2, turns out to have a large impact on performance. If we relax this constraint, our performance on Carberp drops to 39% on reads and 54% on writes for $N = 10$, and our performance on Zeus shows a similar decline (see Table 11). This appears to be caused by two factors. First, functions with very little data all look roughly the same and are likely to be confused with one another by our matching algorithm. Second, including these small reads and writes makes it more likely that incidental reads and writes to the stack such as return addresses and frame pointer; because the data from such reads and writes will change from run to run, this adds noise to the histogram that makes matching more difficult.

5.5 Conclusion and Future Work

In this chapter we demonstrated a new technique for matching functions in unknown malware binaries using *dynamic* features. From our preliminary results, we believe the technique is quite promising and deserving of further study; even with the preliminary nature of our current algorithms and evaluation, our performance exceeds the state of the art in some cases.

The original premise, that different implementations of the same high-level functionality might also be matched, remains an open question. Due to the difficulty of obtaining ground truth in malware samples, we hope to perform an extensive evaluation between two parallel

non-malicious code bases, such as the Linux and FreeBSD kernels or the GNU coreutils and their BSD counterparts. Because functional equivalence is easier to determine in these cases (e.g., the `sort` utility clearly does the same thing in both its GNU and BSD variants; however, it is less clear that any two malware functions are equivalent), we can get an accurate evaluation of how well our chosen features match new implementations of the same functionality.

We also have a number of ideas that may improve the accuracy of the matching. First, we noticed that the highest matches for a given function were its callees and callers. This makes sense: functions will typically pass data they handle to subroutines, so the data signature of many functions will look similar to their neighbors on the callstack. In future work we hope to explore methods of factoring out this similarity, which is not useful for identifying the particular functionality of a given routine, by subtracting out the histograms of its neighbors. The resulting *residue* may then better capture the unique dynamic data handled by that function.

It may also be possible, by incorporating domain knowledge, to come up with *dynamic data signatures* that identify certain classes of functions. For example, a recent paper [126] located functions that implement decryption of DRM-protected media by noting that such functions have inputs that are both highly random and high-entropy, but outputs that are non-random. Similarly, it should be possible to locate decompression routines by looking at the compressibility of input and output data. We hope to explore whether there are other such signatures that can be used to identify unknown malware behaviors in future work.

Our initial results indicate that dynamic signatures may prove much more robust to the types of function changes,such as compiler optimizations, that often baffle static function matching techniques, and we have hope that they may be extended to automatically provide high-level information about unknown implementations as well. Such a capability would have a large impact on patch-based exploit generation, malware analysis, and vulnerability research.

CHAPTER VI

CONCLUSION

In this thesis we have presented a number of novel techniques that uncover high-level semantic information about closed-source programs and operating systems for security. The algorithms presented in Chapters 2–4 have significantly improved the feasibility of virtual machine based intrusion detection and memory forensics by reducing the "semantic gap" between high- and low-level views of vulnerable operating systems. We also presented a new technique that provides the first steps toward a general system for understanding the behavior of unknown programs; moreover, we found that this technique outperformed the state of the art on the simpler task of matching functions between different compiler optimization levels.

We conclude by discussing several large, open problems in both program understanding and intrusion detection that can operate without trusting the OS.

6.0.1 The "Strong" Semantic Gap

In a recent paper surveying virtual machine introspection techniques [48], Jain et al. noted that most work on the semantic gap (including our own) is forced to assume that despite malicious modification, the underlying semantics of the system's data structures remain unchanged. This assumption has been challenged, however, by attacks such as Direct Kernel Structure Manipulation (DKSM) [5], which alters the layout of kernel data structures and thereby invalidates the semantic meaning learned by systems such as those described in Chapters 2 and 3. Jain et al. name this more difficult challenge, of maintaining security monitoring in the face of semantic manipulation attacks, the "strong" semantic gap problem, and conclude that there has been very little research on it.

In the limit, it seems clear that without imposing constraints (either in the threat model or, preferably, through controls enforced by hardware) on modifications malicious code running at the same privilege level as the operating system can make, maintaining

security monitoring is impossible. For example, kernel malware may simply set up its own small operating system and scheduler and share time between it and the main OS; while this may seem farfetched, some proof-of-concept rootkits [2] do go so far as to replace the OS scheduler with their own.

Can automated reverse engineering and program understanding be made to work even in this highly adversarial scenario? That is, could we devise analyses that are both fast and accurate enough to analyze a potentially unknown operating system and extract actionable security information about its behavior? Or, failing that, what is the minimum set of restrictions we need to enforce in order to guarantee that our semantic assumptions cannot be violated?

6.0.2 Understanding Embedded Systems

While understanding *software* has been the focus of this thesis, the proliferation of embedded computing devices poses new challenges. In such systems, simply analyzing the software on the device may be insufficient to evaluate its security and protect it from attack. Instead, we may need to understand how its hardware peripherals work—what the valid ranges of inputs and outputs are, what states it can get into, what attack surface it exposes to the world, and what effect its hardware can have on the physical world (e.g., can your printer actually be forced to catch fire?).

Some of these questions are well beyond the scope of our techniques: it is unlikely we will be able to apply software analysis to reverse engineer the physical properties of hardware, for example. However, it may be possible in some cases to infer certain properties of the hardware through close analysis of the software interfaces used to communicate with it. In some preliminary work not reported in this thesis, we were able to infer constraints on values returned from embedded hardware peripherals by performing symbolic execution [59] on its firmware and solving path constraints that followed a successful (non-crashing) path through the firmware boot process. It remains an open question, however, how much we can discover about hardware in this way—complicated peripherals may have complicated state machines or even be full-fledged computing devices in their own right, which places

fundamental limits on what fully automated techniques can achieve. As with undecidable problems in program analysis, however, a great deal of progress may be achievable with careful application of heuristics and assumptions about common cases.

6.0.3 Reverse Engineering-Friendly Construction

The programs and systems that are the targets of our analyses in this thesis were all designed to be, if not actively hostile (in the case of malware), then at least indifferent to the needs of outsiders wishing to understand their internal details. In addition to developing ever-more powerful ways to tease out the hidden implementation details of such systems, we might instead turn the question around and ask whether we can construct programming languages and tools that expose the internals of a program in a self-documenting way. For example, a compiler might be able to automatically produce parsers for the in-memory data structures used by a program, or generate a list of useful hook points throughout the code, such as places where data crosses marked security boundaries.

In the realm of hardware this is likely to be even more difficult, both because of the lack of standardized tools for hardware design and the reticence of hardware manufactures to give out what they consider trade secrets. Nevertheless we may seek methods that augment the tools used in hardware design so that they produce machine-readable descriptions of the device's functionality and the protocols needed to interact with it. This would provide numerous benefits for security analyses: the descriptions could inform static analyses of the firmware that interacts with such hardware, and dynamic analyses that rely on being able to emulate hardware platforms could potentially automatically generate emulated peripheral models. As a side benefit, having such open hardware descriptions readily available would open up such embedded systems to alternative operating systems, improving user freedom.

Conclusion

The solutions to these problems will determine the privacy and security of the next generation of computing devices. It is vital that we have ways of quickly understanding the attack surface and vulnerabilities of embedded systems that can affect the physical world, be able to quickly reverse engineer novel malware, and provide protection for legacy systems until

they can be replaced by newer and hopefully more secure versions.

Even at the level of the common user, it is important that we have at our disposal robust automated techniques for understanding the software we use. Events such as the discovery of censorship and surveillance in the Chinese version of Skype [60] have shown that reverse engineering can be a way of combating infringements on human rights, and the proliferation of advertising as the primary way of monetizing mobile applications and web sites means that we must be even more vigilant and aware of what private information is exposed to software eager to profile and target us.

If we seek to have control over the computing systems that have increasingly become enmeshed in our lives, we must start by understanding how they work and what they do, and automated program analysis techniques are a vital part of achieving this.

APPENDIX A

SAMPLE TAP POINT CONTENTS FROM TZB

Here, we reproduce a selection of tap points from the same cluster as dmesg and filename tap points.

```
/etc/rc.d/ipfw/etc/rc.d/NETWORKING/etc/rc.d/netwait/etc/rc
.d/mountcritremote/etc/rc.d/devfs/etc/rc.d/ipmon/etc/rc.d/
mdconfig2/etc/rc.d/newsyslog
```

```
r=/sNsnWs/fuiebu/ r=ceremdsecd_t_co_artpachg=tSooadSebaabf
/faa_N_=_peOfA=fA=feTr=tul.n=_eo/.b_Yt_vtectvifat=a=-sd_Ee
Ofu=u_0y:nF:tRseeeeEfciOtmdtuinlrlrrlpp/nppfpcepinl=l=.llN
lNlgllpl_.4l_l_2/l_l_22lileldlylo- 21laltlat=rrrsbgrskgni/
```

```
russian|Russian Users Accounts:  :charset=KOI8-R:        :
lang=ru_RU.KOI8-R:     :       :passwd_format=md5:     :co
pyright=/etc/COPYRIGHT:      :welcome=/etc/motd:     :sete
nv=MAIL=/var/mail/$,BLOCKSIZE=K,FTP_PASSIVE_MODE=YES:
```

```
nss_compat.so.1dhclientShared object ‘‘nss_compat.so.1’’ n
ot found, required by ‘‘dhclient’’nss_nis.so.1dhclientShar
ed object ‘‘nss_nis.so.1’’ not found, required by ‘‘dhclie
nt’’nss_files.so.1dhclientShared object ‘‘nss_files.so.1’’
```

```
digraph geom {
z0xc1d8de00 [shape=box,label=’’PART\nada0\nr#2’’];
z0xc1f4f640 [label=’’r1w0e0’’];
z0xc1f4f640 -> z0xc1e9eb00;
```

```
/sbin/in/bin/sh/bin/stt/sbin/sysctl/bin/ps/sbin/sysctl/sbi
n/rcorde/bin/cat/sbin/md/sbin/sysctl/sbin/sysctl/bin/ken/s
bin/dumpon/bin/ln/bin/ps/sbin/sysctl/sbin/sysctl/sbin/sysc
tl/sbin/sysctl/bin/ps/bin/dd/sbin/sysctl/bin/dat/bin/df/sb
```

```
/boot/kernel/kernel00000000-0000-0000-0000-000000000000000
00000-0000-0000-0000-0000000000000000000-0000-0000-0000-0
00000000000993c915d-3e9f-11e2-a557-525400123456993c915d-3e
9f-11e2-a557-525400123456/boot/kernel/kernel/boot/kernel/k
...
modulesoptions       CONFIG_AUTOGENERATED
ident   GENERIC
machine i386
cpu     I686_CPU
cpu     I586_CPU
cpu     I486_CPU

<mesh>
  <class id=''0xc10362c0''>
    <name>FD</name>
  </class>
  <class id=''0xc1009a80''>

#!/bin/sh
#
# $FreeBSD: release/9.0.0/etc/rc.d/newsyslog 197947 2009-1
0-10 22:17:03Z dougb $
...
set_rcvar()
{
        case $# in
        0)
                echo ${name}_enable
                ;;
        1)
```

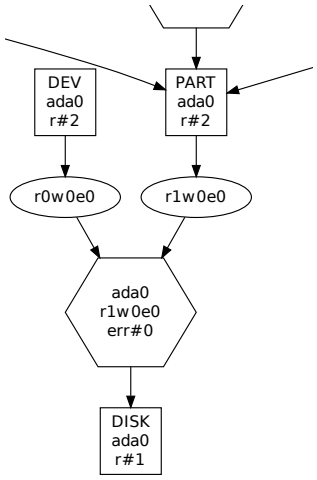


Figure 28: Detail from rendering of Graphviz file captured from a FreeBSD boot tap point, apparently depicting disk geometry

REFERENCES

[1] "Raspberry Pi: An ARM GNU/Linux box for $25." `http://www.raspberrypi.org/`.

[2] 90210, "Bypassing Klister 0.4 with no hooks or running a controlled thread scheduler." `https://www.rootkit.com/newsread.php?newsid=235`.

[3] Anonymous, "Zeus source code on GitHub." `https://github.com/Visgean/Zeus`, 2011.

[4] Arthur, D. and Vassilvitskii, S., "k-means++: the advantages of careful seeding," in *Proceedings of the ACM-SIAM symposium on Discrete algorithms*, 2007.

[5] Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., and Xu, D., "DKSM: Subverting virtual machine introspection for fun and profit," *IEEE Symposium on Reliable Distributed Systems*, 2010.

[6] Bailey, M., Oberheide, J., Andersen, J., Mao, Z., Jahanian, F., and Nazario, J., "Automated classification and analysis of internet malware," in *Recent Advances in Intrusion Detection* (Kruegel, C., Lippmann, R., and Clark, A., eds.), vol. 4637 of *Lecture Notes in Computer Science*, pp. 178–197, Springer Berlin Heidelberg, 2007.

[7] Baliga, A., Ganapathy, V., and Iftode, L., "Automatic inference and enforcement of kernel data structure invariants," in *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*, (Anaheim, California, USA), 2008.

[8] Banerjee, A., Merugu, S., Dhillon, I. S., and Ghosh, J., "Clustering with Bregman divergences," *J. Mach. Learn. Res.*, vol. 6, Dec. 2005.

[9] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *ACM Symposium on Operating System Principles (SOSP)*, 2003.

[10] Bayer, U., Milani Comparetti, P., Hlauscheck, C., Kruegel, C., and Kirda, E., "Scalable, Behavior-Based Malware Clustering," in *Network and Distributed System Security Symposium (NDSS)*, 2009.

[11] Bell, D. E. and LaPadula, L. J., "Secure computer systems: Mathematical foundations," Technical Report MTR-2547, MITRE Corp., Bedford, MA, 1973.

[12] Bellard, F., "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005.

[13] Betz, C., "MemParser." `http://sourceforge.net/projects/memparser`.

[14] Bird, S., Klein, E., and Loper, E., *Natural Language Processing with Python*. O'Reilly Media, 2009.

[15] bugcheck, "GREPEXEC: Grepping executive objects from pool memory," *Uninformed Journal*, vol. 4, 2006.

[16] Bursztein, E., Hamburg, M., Lagarenne, J., and Boneh, D., "OpenConflict: Preventing real time map hacks in online games," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[17] Butler, J., "FU rootkit." `http://www.rootkit.com/project.php?id=12`.

[18] Caballero, J., Johnson, N. M., McCamant, S., and Song, D., "Binary code extraction and interface identification for security applications," in *Network and Distributed Systems Security Symposium (NDSS)*, (San Diego, CA), 2010.

[19] Caballero, J., Yin, H., Liang, Z., and Song, D., "Polyglot: automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the ACM conference on Computer and communications security*, 2007.

[20] Checkoway, S., Feldman, A. J., Kantor, B., Halderman, J. A., Felten, E. W., and Shacham, H., "Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage," in *Electronic Voting Technology / Workshop on Trustworthy Elections (EVT/WOTE)*, (Montreal, Canada), 2009.

[21] Chilimbi, T. M., Davidson, B., and Larus, J. R., "Cache-conscious structure definition," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI)*, (Atlanta, GA), 1999.

[22] Chipounov, V., Kuznetsov, V., and Candea, G., "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, 2011.

[23] Chiueh, T. and Hsu, F., "RAD: a compile-time solution to buffer overflow attacks," in *International Conference on Distributed Computing Systems*, 2001.

[24] Christodorescu, M., Jha, S., Seshia, S., Song, D., and Bryant, R., "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*, pp. 32–46, May 2005.

[25] Christodorescu, M. and Jha, S., "Testing malware detectors," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, (Boston, MA), 2004.

[26] Comparetti, P., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., and Zanero, S., "Identifying dormant functionality in malware programs," in *IEEE Symposium on Security and Privacy*, pp. 61–76, May 2010.

[27] Cozzie, A., Stratton, F., Xue, H., and King, S. T., "Digging for data structures," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[28] Cui, W., Kannan, J., and Wang, H. J., "Discoverer: automatic protocol reverse engineering from network traces," in *Proceedings of the USENIX Security Symposium*, 2007.

[29] Cui, W., Peinado, M., Chen, K., Wang, H. J., and Irun-Briz, L., "Tupni: automatic reverse engineering of input formats," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

[30] Detristan, T., Ulenspiegel, T., Malcom, Y., and von Underduk, M. S., "Polymorphic shellcode engine using spectrum analysis." `http://www.phrack.com/issues.html?issue=61&id=9`, 2003.

[31] Dinaburg, A., Royal, P., Sharif, M., and Lee, W., "Ether: malware analysis via hardware virtualization extensions," in *ACM Computer and Communications Security*, (Alexandria, VA), 2008.

[32] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W., "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2011.

[33] Dolan-Gavitt, B., Srivastava, A., Traynor, P., and Giffin, J., "Robust signatures for kernel data structures," in *ACM Computer and Communications Security (CCS)*, 2009.

[34] Egele, M., Woo, M., Chapman, P., and Brumley, D., "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), USENIX Association, Aug. 2014.

[35] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C., "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, 2007.

[36] F-Secure, "Rootkit:W32/HacDef description." `http://www.f-secure.com/v-descs/hacdef.shtml`.

[37] F-Secure, "A different twist on the path to the kernel." `http://www.f-secure.com/weblog/archives/00001507.html`, 2008.

[38] Fogla, P. and Lee, W., "Evading network anomaly detection systems: formal reasoning and practical techniques," in *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*, (Alexandria, VA), 2006.

[39] Forrester, J. E. and Miller, B. P., "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium (WSS)*, (Seattle, WA), 2000.

[40] Fu, Y. and Lin, Z., "Space traveling across VM: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.

[41] Garfinkel, T., "Traps and pitfalls: Practical problems in in system call interposition based security tools," in *Network and Distributed Systems Security Symposium (NDSS)*, (San Diego, CA), 2003.

[42] Garfinkel, T. and Rosenblum, M., "A virtual machine introspection based architecture for intrusion detection," in *Network and Distributed Systems Security Symposium (NDSS)*, 2003.

[43] Gegeny, J., "Decrypting Carberp C&C communication." `http://securityblog.s21sec.com/2011/07/decrypting-carberp-c-communication.html`, 2011.

[44] Gilbert, K., "Hurricane sandy serves as lure to deliver Sykipot." `http://securityblog.verizonbusiness.com/2012/10/31/hurricane-sandy-serves-as-lure-to-deliver-sykipot/`.

[45] Gundy, M. V., Chen, H., Su, Z., and Vigna, G., "Feature omission vulnerabilities: Thwarting signature generation for polymorphic worms.," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, (Miami Beach, FL), 2007.

[46] "Haiku OS project." `http://haiku-os.org/`.

[47] Hubert, L. and Arabie, P., "Comparing partitions," *Journal of Classification*, vol. 2, no. 1, 1985.

[48] Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R., "SoK: Introspections on trust and the semantic gap," in *IEEE Symposium on Security and Privacy*, (San Jose, CA), May 2014.

[49] Jang, J., Brumley, D., and Venkataraman, S., "BitShred: Feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 309–320, ACM, 2011.

[50] Jang, J., Woo, M., and Brumley, D., "Towards automatic software lineage inference," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, (Washington, D.C.), pp. 81–96, USENIX, 2013.

[51] Jarmoc, J., "SSL/TLS interception proxies and transitive trust," in *Black Hat Europe*, March 2012.

[52] Jiang, X., Xu, D., and Wang, X., "Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction," in *ACM Computer and Communications Security (CCS)*, (Alexandria, VA), 2007.

[53] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Antfarm: tracking processes in a virtual machine environment," in *Proceedings of the USENIX Annual Technical Conference (ATEC)*, (Boston, MA), 2006.

[54] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "VMM-based hidden process detection and identification using lycosid," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)*, (Seattle, WA), 2008.

[55] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M., "Detecting past and present intrusions through vulnerability-specific predicates," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[56] Kálnai, P. and Hořejší, J., "Dissecting banking trojan Carberp." `http://www.rsaconference.com/writable/presentations/file_upload/ht-t06-dissecting-banking-trojan-carberp_copy1.pdf`, 2013.

[57] Kephart, J. and Arnold, W., "Automatic extraction of computer virus signatures," in *Proceedings of the 4th International Virus Bulletin Conference (VB)*, (Jersey, Channel Islands, UK), 1994.

[58] Kim, H.-A. and Karp, B., "Autograph: Toward automated, distributed worm signature detection," in *Proceedings of the 13th conference on USENIX Security Symposium*, vol. 13, (San Diego, CA), 2004.

[59] King, J. C., "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 385–394, July 1976.

[60] Knockel, J., Crandall, J. R., and Saia, J., "Three Researchers, Five Conjectures: An Empirical Analysis of TOM-Skype Censorship and Surveillance," in *Free and Open Communications on the Internet*, (San Francisco, CA, USA), USENIX, 2011.

[61] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., and Wang, X., "Effective and efficient malware detection at the end host," in *USENIX Security Symposium*, (Montreal, Canada), 2009.

[62] Kolbitsch, C., Holz, T., Kruegel, C., and Kirda, E., "Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries," in *IEEE Symposium on Security and Privacy*, 2010.

[63] Korel, B. and Laski, J., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, 1988.

[64] Krebs, B., "Carberp code leak stokes copycat fears." `http://krebsonsecurity.com/tag/carberp-source-code-leak/`, 2013.

[65] Kreibich, C. and Crowcroft, J., "Honeycomb: creating intrusion detection signatures using honeypots," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, 2004.

[66] Kruse, P., "Complete ZeuS sourcecode has been leaked to the masses." `http://www.csis.dk/en/csis/blog/3229/`.

[67] Kullback, S. and Leibler, R. A., "On information and sufficiency," *Annals of Mathematical Statistics*, vol. 22, 1951.

[68] Lanzi, A., Sharif, M. I., and Lee, W., "K-tracer: A system for extracting kernel malware behavior," in *Network and Distributed Systems Security Symposium (NDSS)*, 2009.

[69] Laureano, M., Maziero, C., and Jamnhour, E., "Intrusion detection in virtual machine environments," in *Euromicro Conference*, 2004.

[70] Lee, J., Avgerinos, T., and Brumley, D., "TIE: Principled reverse engineering of types in binary programs," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), 2011.

[71] Lee, W., Stolfo, S. J., and Chan, P. K., "Learning patterns from UNIX process execution traces for intrusion detection," in *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pp. 50–56, 1997.

[72] Levenshtein, V., "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics - Doklady*, vol. 10, 1966.

[73] Li, Z., Sanghi, M., Chen, Y., Kao, M.-Y., and Chavez, B., "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, (Oakland, CA), 2006.

[74] Lin, J., "Divergence measures based on the Shannon entropy," *IEEE Trans. Inf. Theor.*, vol. 37, Sept. 2006.

[75] Lin, Z., Jiang, X., Xu, D., and Zhang, X., "Automatic protocol format reverse engineering through context-aware monitored execution," in *Network and Distributed Systems Symposium*, 2008.

[76] Lin, Z. and Zhang, X., "Deriving input syntactic structure from execution," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[77] Lin, Z., Zhang, X., and Xu, D., "Automatic reverse engineering of data structures from binary execution," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), 2010.

[78] Linn, C. and Debray, S., "Obfuscation of executable code to improve resistance to static disassembly," in *ACM Computer and Communications Security (CCS)*, 2003.

[79] Lui, M. and Baldwin, T., "Langid.Py: An off-the-shelf language identification tool," in *Proceedings of the ACL 2012 System Demonstrations*, ACL '12, (Stroudsburg, PA, USA), Association for Computational Linguistics, 2012.

[80] mathfigure, "ICU64: Real-time hacking of a C64 emulator."

[81] Microsoft Corp., "Debug interface access SDK." `http://msdn.microsoft.com/en-us/library/x93ctkx8.aspx`.

[82] Microsoft Corporation, "EvtQuery function." `http://msdn.microsoft.com/en-us/library/windows/desktop/aa385466(v=vs.85).aspx`.

[83] Microsoft Corporation, "RtlAllocateHeap on MSDN." `http://msdn.microsoft.com/en-us/library/ff552108(VS.85).aspx`.

[84] Microsoft Corporation, "Windows research kernel." `http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx`.

[85] Miller, B. P., Fredriksen, L., and So, B., "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, 1990.

[86] Miller, B. P., Koski, D., Pheow, C., and Maganty, L. V., "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," tech. rep., 1995.

[87] Mission Critical Linux, "Core analysis suite (crash)." `http://www.missioncriticallinux.com/projects/crash/`.

[88] Moser, A., Kruegel, C., and Kirda, E., "Exploring multiple execution paths for malware analysis," in *IEEE Symposium on Security and Privacy*, pp. 231–245, May 2007.

[89] Müller, T., Freiling, F. C., and Dewald, A., "TRESOR runs encryption securely outside RAM," in *Proceedings of the 20th USENIX conference on Security*, 2011.

[90] Narayanasamy, S., Pokam, G., and Calder, B., "Bugnet: Continuously recording program execution for deterministic replay debugging," in *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA)*, (Washington, DC, USA), 2005.

[91] National Institute of Standards and Technology (NIST), "The CFReDS project." `http://www.cfreds.nist.gov/`.

[92] Newsome, J., Karp, B., and Song, D., "Polygraph: Automatically generating signatures for polymorphic worms," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, (Oakland, CA), 2005.

[93] Oehlert, P., "Violating assumptions with fuzzing," *IEEE Security and Privacy*, vol. 3, no. 2, 2005.

[94] Payne, B. D., Carbone, M., and Lee, W., "Secure and flexible monitoring of virtual machines," in *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[95] Payne, B. D., Carbone, M., Sharif, M., and Lee, W., "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy*, 2008.

[96] Petroni, Jr., N. L., Fraser, T., Molina, J., and Arbaugh, W. A., "Copilot - a coprocessor-based kernel runtime integrity monitor," in *USENIX Security Symposium*, 2004.

[97] Petroni, Jr., N. L. and Hicks, M., "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, (Alexandria, VA), 2007.

[98] Polishchuk, M., Liblit, B., and Schulze, C. W., "Dynamic heap type inference for program understanding and debugging," *SIGPLAN Not.*, vol. 42, no. 1, pp. 39–46, 2007.

[99] Quarkslab, "iMessage privacy." `http://blog.quarkslab.com/imessage-privacy.html`, 2013.

[100] Roesch, M., "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX conference on System administration (LISA)*, (Seattle, WA), 1999.

[101] Rumsfeld, D., "DoD news briefing - Secretary Rumsfeld and Gen. Myers." February 2002.

[102] Rutkowska, J., "Klister v0.3." `https://www.rootkit.com/newsread.php?newsid=51`.

[103] Rutkowska, J., "modGREPER - hidden kernel modules detector." `https://www.rootkit.com/newsread.php?newsid=315`, 2005.

[104] Schuster, A., "Searching for processes and threads in Microsoft Windows memory dumps," in *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS)*, (Lafayette, IN), 2006.

[105] Seshadri, A., Luk, M., Qu, N., and Perrig, A., "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of 21st ACM SIGOPS symposium on Operating systems principles (SOSP)*, (Stevenson, WA), 2007.

[106] Shacham, H., "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *ACM Computer and Communications Security (CCS)*, (Alexandria, VA), 2007.

[107] Singh, S., Estan, C., Varghese, G., and Savage, S., "Automated worm fingerprinting," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (San Francisco, CA), 2004.

[108] Sinnadurai, S., Zhao, Q., and Wong, W., "Transparent runtime shadow stack: Protection against malicious return address modifications." `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702`, 2008.

[109] Slowinska, A., Stancescu, T., and Bos, H., "Towards precise data structure recognition in stripped binaries," in *Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), 2011.

[110] Slowinska, A., Stancescu, T., and Bos, H., "Howard: A dynamic excavator for reverse engineering data structures," in *Network and Distributed Systems Symposium*, 2011.

[111] Solar Eclipse, "Tiny PE." `http://www.phreedom.org/solar/code/tinype/`, 2006.

[112] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P., "BitBlaze: A new approach to computer security via binary analysis," in *Information systems security*, 2008.

[113] Song, Y., Locasto, M., Stavrou, A., Keromytis, A., and Stolfo, S., "On the infeasibility of modeling polymorphic shellcode," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, (Alexandria, VA), 2007.

[114] Spinellis, D., "Reliable identification of bounded-length viruses is NP-complete," *IEEE Transactions on Information Theory*, vol. 49, no. 1, 2003.

[115] Srinivasan, D., Wang, Z., Jiang, X., and Xu, D., "Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring," in *Proceedings of the ACM conference on Computer and communications security*, 2011.

[116] Srivastava, A. and Giffin, J., "Tamper-resistant, application-aware blocking of malicious network connections," *Recent Advances in Intrusion Detection*, 2008.

[117] Steinhaus, H., "Sur la division des corp materiels en parties," *Bull. Acad. Polon. Sci*, vol. 1, 1956.

[118] Symantec Corp., "Trojan.Zbot." `http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99`, 2010.

[119] Ször, P. and Ferrie, P., "Hunting for metamorphic," in *Proceedings of the 11th International Virus Bulletin Conference*, (Prague, Czech Republic), 2001.

[120] valerino, "Please don't greap me!." `https://www.rootkit.com/newsread.php?newsid=316`, 2005.

[121] Vasudevan, A. and Yerraballi, R., "Stealth breakpoints," in *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, (Tucson, AZ), 2005.

[122] Vendicator, "Stack shield: A "stack smashing" technique protection tool for Linux." `http://www.angelfire.com/sk/stackshield/`.

[123] VMWare, "VMWare Server." `http://www.vmware.com/products/server/`.

[124] Walters, A., "The Volatility framework: Volatile memory artifact extraction utility framework." `https://www.volatilesystems.com/default/volatility`.

[125] Walters, A. and Petroni, N., "Volatools: Integrating volatile memory forensics into the digital investigation process," in *Blackhat Federal*, (Washington, DC), 2007.

[126] Wang, R., Shoshitaishvili, Y., Kruegel, C., and Vigna, G., "Steal this movie: Automatically bypassing drm protection in streaming media services," in *USENIX Security Symposium*, (Washington, D.C.), pp. 687–702, USENIX, 2013.

[127] Wang, Z., Jiang, X., Cui, W., and Wang, X., "Countering persistent kernel rootkits through systematic hook discovery," in *Recent Advances in Intrusion Detection*, 2008.

[128] Willems, C., Holz, T., and Freiling, F., "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security & Privacy*, vol. 5, March 2007.

[129] Yan, Z., "perf, x86: Haswell LBR call stack support." `http://lwn.net/Articles/535152/`.